Design Patterns in Learning to Program

by

Ron Porter, *B.A., Graduate Diploma in Computer Science, B.Sc.(Comp.Sc)(Hons)*
School of Informatics and Engineering,
Faculty of Science and Engineering

November 24, 2006

A thesis presented to the
Flinders University of South Australia
in total fulfillment of the requirements for the degree of
Doctor of Philosophy

Adelaide, South Australia, 2006
© (Ron Porter, 2006)

Table of Contents

List of Figures

List of Tables

Abstract

This thesis argues the case for the use of a pattern language based on the basic features of the programming language used in instruction for the teaching of programming. We believe that the difficulties that novices are known to have encountered with the task of learning to program ever since the inception of computers derive from a basic misfit between the language used to communicate with a computer, the programming language, and the way that humans think. The thrust of the pattern language idea is that patterns are the essential element in understanding how the mind words in that they are the source of that relationship that we call 'meaning'. What an entity or event 'means' to us derives from the effect that it has on us as living biological beings, a relationship that exists in the 'real world', not from any linguistic relationship at the symbolic level. Meaning, as a real world relationship, derives from the patterns of interactions that constitute being. The meaning that an entity has for an individual is more than can be expressed in a formal definition, definitions are matters of agreement, convention, not the pattern of experience that the individual has acquired through living. What is missing for a novice in any skill acquisition process is meaning, the pattern of experience. All that we can give them using a formal linguistic system like a programming language is definitions, not meaning. Pattern language is the way that we think because it exists at that fundamental level of experience as living beings. The patterns of experience become the patterns of thought through recurrence, not through definition. But this takes time, so in presenting new material to a person trying to learn, we have to present it in the form of a pattern language, the "cognitive map" that drives the problem soving process. Creativity is *always* a function of combining ideas, what is really being created is new meaning, not a program, or a house, or a poem, or a sculpture - these things are mere implementations of meaning. Ultimately meaning can derive only from experience, the pattern of life around us, so creativity is the language of experience, pattern language. The mind is the product of experience, creativity its modus operandi.

Certification

I certify that this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

As requested under Clause 14 of Appendix D of the *Flinders University Research Higher Degree Student Information Manual* I hereby agree to waive the conditions referred to in Clause 13(b) and (c), and thus

- Flinders University may lend this thesis to other institutions or individuals for the purpose of scholarly research;
- Flinders University may reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signed Dated

Ron Porter

Acknowledgements

As with anything else in life, producing a dissertation involves the use of a pattern language, and one of the patterns in that language is to reflect, at the very end of the journey, on those who have been, knowingly or inadvertantly, fellow travellers, to explore, as it were, the patterns of human connections that make you who you are and the thesis what it is.

My original and enduring inspiration is Lel, who is responsible in ways that she can never know, for this journey, and who, to this day, still whispers in my ear whenever I need reminding of some pattern I have forgotten.

In the patterns of everyday life, family and friends make important contributions of which they are mostly unaware, but which are essential in making any journey like this one possible, and a few friends who deserve particular mention are Sok Kiang Lee, Gisela Bogdan, Lochie and Sam Davies, Jayne White for the insights she gave me into educational and curriculum issues, and, latterly Morgan and Jackie Stringer.

This probable lack of awareness of their vital contribution extends also to many of one's colleagues. Two such contributors that deserve special mention are Denise de Vries and Aaron Ceglar whose input is incalculable. Most of the material on the "epistemic cut" was honed in discussion with the latter, who, much to his own surprise, turned out to be the most philosophically aware of my colleagues, save only for my erstwhile room-mate, Scott Vallance, who was also the source of many insights.

A more conscious contribution was made by Tiffany Winn and Lorraine Harker, fellow travellers on the pattern language route. Tiffany, in particular, opened many doors in the interesting discussions we had along the way.

Through one of those doors stepped Jim Coplien, who as well as contributing to this dissertation directly, introduced me, somewhat unconventionally, to Joe Bergin, whose idea it was to attempt to experimentally measure the effectiveness of patterns in programming pedagogy.

Of course, the most vital contribution is made by one's supervisor, and in in my case I chose wisely. Paul Calder was the most constant of my fellow travellers, and although some of the places I wanted to visit were, to say the least of it, a bit 'weird', he was always there for the ride. His guidance was the most significant input into this journey.

Many others contributed through their presence in the structure of academic institutions like this one, and my thanks go to all who provide the means for such explorations of unfamiliar territory.

Ron Porter
February 2003
Adelaide.

Preface - Scope and Outline

The thrust of this dissertation is that learning to program, indeed learning any skill, involves the use of a pattern language either explicitly or implicitly. This is because any skill depends on the conceptual structure in the mind, a cognitive map, as this is the only way that creative potential can be expressed. However, there is actually nothing new in this, as humans we spend our lives from day one building such knowledge structures in our mind in every field of endeavour, a process that can only be based on patterns of experience, the things that are the same for all of us, and the connections between them, in short, a pattern language. The trouble is that we do this in a largely unconscious fashion, such as in the way that we 'pick up' the grammar of our first spoken language. This unconscious development of cognitive structure, however, becomes inefficient in fields of great complexity or where practice is difficult without a reasonably developed cognitive structure already in place. Programming is such a field because it is based on the strict symbolic logic of the machine which is not familiar from the everyday cut and thrust of life experience that drives many other disciplines, and so there is no obvious pre-existing relevant knowledge structure.

In other words, we cannot rely on the 'natural' resonance of experience in the programming field with everyday experience as can be done, to some extent, in other fields. Therefore the aim of this thesis is to demonstrate how an evolving pattern language can provide the sort of resonances that take advantage of the connections that already exist in the mind, missing in instruction based on pure logic. This is not an entirely unique problem, many fields such as mathematics, philosophy, science and so on, indeed reasoning in general, involve elements of logical thinking. So, it is possible to find, in other areas, methods of driving the acquisition of creative skill that illustrate the power of the pattern process that we are advocating. Therefore, we touch on many strands which may appear at first to be only loosely connected. This is true, in some respects, but the factor common to all of them is that they demonstrate, in some way, the fundamental correspondence between evolution as design for life through patterns of experience and learning as design for life through patterns of experience. Just as evolution involves a codification of experience in the strict 'logic' of DNA sequence, so too does programming involve a codification of experience in the strict 'logic' of machine instruction sequence. It follows, therefore, that cognitive development, learning in short, as design for life through patterns of experience, should provide the means to enable people to learn to program.

So what we are attempting to show here is that pattern language, as an explicit rather than implicit factor in instruction, is the way forward in crossing the gap between general thinking based on everyday experience and that required for dealing with a strict symbolic logic. This attempt necessarily involves exploring many areas of knowledge to illustrate our argument, and from this exploration it is clear that the cognitive structure involved in any task must reflect the objective conceptual structure (the pattern language) of the field concerned, and this applies in every human skill. While we concentrate our attention in the programming domain on the imperative-procedural-OO paradigm, we do this for the sake of clarity, not because the pattern process applies only here. Ours is a philosophical task, an exploration of the issues involved in acquiring a skill, not a technical one, so, inevitably, the dissertation involves a conceptual (philosophical), even a narrative, flow rather than a logical (technical) one. That such a non-technical and non-reductionist approach is necessary is shown by the moral core of the pattern language idea, the proposition that order and coherence are properties of the whole, not the parts of which it is made.

In Chapter 1 we establish the scope of the issue dealt with in this dissertation - the use of pattern languages to address the problems that novices exhibit in learning to program - and the thrust of our argument is that it is the difference between how humans think and the logical rigor and mechanical nature of programming languages that lies at the heart of the matter. As Christopher Alexander's pattern language concept is directed at the process of designing solutions to problems in the 'real world', we introduce it as a means of providing a cognitive map for novices in the programming domain.

Chapter 2 provides an overview of prior work in the fields opened up for discussion in the introduction in an attempt to provide a broad philosophical basis for the project. Although the idea of a language of patterns is relatively new, patterns themselves have long been recognised as a significant factor in human thinking processes. We examine their use in terms of the learning process and educational practice, and, more particularly, in introductory programming where we find that ideas such as "chunks" and "schemas" which resonate with the pattern concept predate the explicit use of patterns in programming pedagogy. The main aim of this chapter is to introduce pattern theory in both its classical and Alexandrian forms so that we can set the context for its place in pedagogical theory.

As designing any artefact, including a program, is a creative act, we explore, in Chapter 3, the roots of creativity in everyday human experience. Again we are attempting to establish the problem in its widest context, so the discussion here is mostly philosophical in spirit, leaving the psychological and educational aspects to be dealt with elsewhere.

Chapter 4 argues that the source of the difficulties exhibited by novices stems from the fact that instead of attempting to adjust our pedagogies to better fit how the mind actually works, we have persisted in trying to modify the mind to fit the programming system, to 'make people think like computers', in effect.

A designed artefact of any complexity requires generativity, the combination of multiple concepts to a unity of purpose, and Chapter 5 attempts to demonstrate that the fundamental function of language is generation not communication. Before we can communicate any complex idea, we have to have generated it out of the simpler ideas that constitute its components. Language-as-conceptual-understanding, therefore, is the factor that underlies the human condition, that drives all creativity.

Having explored the role of patterns and of language in human thinking separately, we attempt, in Chapter 6, a synthesis based on the pattern language concept developed by Christopher Alexander. Here we are concerned with the use of pattern languages in education, so our examination is mainly confined to the pedagogical implications of Alexander's theory, which means that there are significant differences with pattern practice as it has developed in software engineering.

In particular, we emphasise the use of pattern language diagrams, an aspect of Alexander's thinking that has not caught on in the software field, and Chapter 7 uses a simple programming exercise as a work-through to demonstrate how the language generates the solution. This is not a logical or mechanical process, the programmer still needs to make creative decisions along the way, but the pattern language diagram clarifies the points at which such decisions are needed and the arrows between the nodes in the network point to the various options available at each of these junctures. As the purpose of our example is to illuminate the actual process involved in using a pattern language to generate the design of a program, we have, of necessity, had to simplify - almost to the point of absurdity in terms of their own domains - all aspects, the patterns, the language, and the programming example itself. But our purpose is not to analyse these aspects, but to illuminate the complex interaction between them. So everything in this chapter is constrained by the necessity for clarity in exposing the *dynamics of process*, an undertaking that we found to be almost impossible in fact.

Because programming is an activity that occurs in the human mind there are psychological implications, and these are the subject of Chapter 8. Here we are concerned with how novices acquire meaning, so our examination is mainly concerned with the problem of *meaning*, what it means for an element of thought to mean something, as this, we believe, is the major implication of Alexander's ideas - it is always meaning, conceptual order, that is being designed, no matter which 'material' domain is involved. This is somewhat of a divergence from the more conventional psychological investigations of programming undertaken elsewhere in the literature, particularly by the Psychology of Programming Interest Group, but our investigation, as always, is coloured by its concentration on the development of what might be termed the 'programming mind.' Most psychological research is driven by empirical concerns that fail to address our fundamentally philosophical approach.

Again, in Chapter 9, our investigation of the psychology of learning to program, we diverge somewhat from more conventional treatments, as we base our analysis on the process of developing expertise in general. The case study of the apparently 'trivial pursuit' of reciting large sequences of digits from memory demonstrates how a pattern language is developed and used to drive expert, even world record, performance of a task.

Chapter 10 outlines the three attempts we made to measure the effectiveness of using pattern languages in teaching people how to program. Although our experiments foundered on the difficulty of balancing the motivation of attendance at 'special' pattern sessions and the need of students to maintain progress in their normal non-pattern programming coursework, we felt that our efforts were of value, both in directing any future attempts, and in illustrating the difficulties involved in measuring performance in any mental activity such as programming.

Any investigation that attempts to cover as much ground as we do in this dissertation, inevitably leaves many loose ends dangling in the breeze. The main task of the conclusion is therefore to gather as many of the threads together as possible, and to demonstrate that the unifying principle is the correspondence between pattern language as a basis for programming-as-design and the functioning of evolution in the derivation of complex natural form, evolution-as-design. Like evolution, programming proceeds on the basis of patterns of experience, not the logical progression of formal symbolic systems such as DNA or programming languages, and it is pattern language rather than logical rigor that drives it.

Chapter 1

Introduction

As teachers of programming we should try to blend the technology of the scientist with the pretence of the craftsman. Sticking to the technology of the scientist means being as explicit as we possibly can about as many aspects of our trade as we can. Now the teaching of programming comprises the teaching of facts - facts about systems, machines, programming languages etc. - and it is very easy to be explicit about them, but the trouble is that these facts represent about 10 percent of what has to be taught: the remaining 90 percent is problem solving and how to avoid unmastered complexity, in short: it is the teaching of thinking, no more and no less. The explicit teaching of thinking is no trivial task, but who said that the teaching of programming is? In our terminology, the more explicitly thinking is taught, the more of a scientist the programmer will become.

Edsger W. Dijkstra (Dijkstra 1982, p. 107)

Meaningful (as opposed to rote) human learning occurs when new knowledge is consciously and purposively linked to an existing framework of prior knowledge in a non-arbitrary, substantive fashion. In rote (or memorized) learning, new concepts are added to the learner's framework in an arbitrary and verbatim way, producing a weak and unstable structure that quickly degenerates. The result of meaningful learning is a change in the way individuals experience the world; a conceptual change.

Michael Zeilik (Zeilik n.d.)

1.1 Why is Programming Difficult?

The problem addressed by this dissertation is the difficulty that people encounter in the task of learning to program a computer. In the current environment, the task of "teaching" people programming falls mainly to three instructional settings.

Some programming instruction takes place at the level of secondary education, and there have been some experimental forays into presenting it at the primary level as well. Nevertheless, it would be reasonably safe to say that most people who learn to program do so after they have left school, that is, they are adults, or close to it. So most instruction takes place at either the tertiary level of education or within the information industry itself. What this tells us is that most, if not all, novice programmers are reasonably highly educated in terms of the population as a whole, and if a major goal of education is to produce individuals who are ready to engage in a productive life in a modern society, *rational agents*, then it is not unreasonable to expect them to be able to be trained for virtually any middle-level career in that society, such as programmer.

The usual explanation for the difficulty we have in training these people is the supposed inherent complexity of the programming task, but this tells us nothing if we don't understand the source and nature of its complexity. So the first aspect we need to analyse here is the machine that is being programmed, to see if the complexity resides there. Any such analysis quickly reveals that although modern computers are wickedly complex in terms of electrical engineering, data paths, logic circuits, and the like, at the level at which they are programmed - basically 'told' what to do - they are, if anything, amazingly simple. A computer has about the same order of complexity at its basic instruction level as the genetic system does in biology.

> It is now known that even humans are produced by the interactions of certain genetic systems and that the generative rules are relatively simple in comparison to the complexity of the end product. This is now accepted as a part of biology, but for most people it is not emotionally real. It's just too incredible. And I think the reason is that we have not yet succeeded in simulating the process.
>
> (Alexander quoted in (Grabow) 1983, p. 47)

Moreover, the biological analogy provides us with a vital clue. The complexity in both systems lies in *what is produced by the code*, or rather, in the programming case, what *can* be produced, not the coding system itself. So if it is true that the programming task is complex, then the complexity involved is a function of the procedures we attempt to automate, the artefacts we try to build with the system, not the programming system itself.

There is an interesting paradox here, in terms of the complexity of the programming task, and that is that the evolution of programming has, in fact, made the substrate *more*, rather than *less* complex. By this we mean that modern programming languages make many more operations available to the programmer than those provided by the base machine on which they run - complexity has been added to the programming system via the compiler. Indeed, this is a case of complexity produced by the code - but deliberately in order to make the concepts used in programming closer to those used in everyday thinking. Moreover, this points to the danger of making a direct connection between complexity and difficulty, the two words refer to different aspects of reality. Complexity, properly considered, is a lack of symmetry or symmetry breaking, it means that there is, in principle, no way that a single aspect of a complex entity can be used to predict, actually or statistically, the properties of other parts - it says, in effect, that there are no real patterns in the entity (Heylighen 1996). Difficulty *can* be the result of complexity, of course, but more usually it is the result of failing to perceive patterns in an entity that actually does have some symmetry. So in our case the programming system is *more* complex than its base, only in the sense that it has more features. But, supposedly at least, it is *less* difficult to use, because those features map more closely to human experience than the base operations do.

This implies that there is something of a dichotomy involved in the development of programming systems. When the designers of a programming language are enjoined to make them as "simple" as possible, "so that a programmer can readily learn and remember all its features, can select the best facility for each of his purposes, can fully understand the effects and consequences of each decision and can then concentrate the major part of his intellectual effort to understanding his problem and his programs rather than his tool" (Hoare 1973, p. 5) they are being asked to fulfill two somewhat contradictory aims. Simplicity in terms of the feature set does not, in itself, make the task of understanding the problem, that is, modelling the problem domain, easier, and it is not even entirely clear that it makes understanding programs easier, although this is intuitively more likely. However, even if, as Backus suggests, most modern programming languages have become so baroque that no simple conceptual model of them exists (Backus 1978), this is not a problem unless you believe that programming and coding are synonymous.

The real force of Hoare's, and Backus' comments is, surely, that the programming system, as a whole, is *more* than the "language" used for implementing a solution. As Peter Naur indicates, "programming properly should be regarded as an activity by which programmers form or achieve a certain kind of insight, a theory, of the matters at hand ... in contrast to what appears to be a more common notion, that programming should be regarded as the production of a program" (Naur 1985, p. 253). Programming then, in this view, consists of two phases, designing a *conceptual* solution and translating the conceptual solution into code form. It should not be surprising, therefore, that a "language" that is supposed to inform both tasks becomes somewhat "baroque" over time. If anything, it is the fact that modern programming languages are being used to perform on these two, quite contradictory, levels that causes them to tend to ornateness. Modern programming languages are a product of the realisation that bare machine operations do not map onto real world activities naturally, which is why it can be said that "the *mode of thinking* [emphasis added] a programming language induces, its paradigm, is more important and has more lasing value than the language itself" (Bal & Grune 1994, p. 1). It might be argued, therefore, that the *real* power of computing lies in reducing the difficulty of doing the tasks that are automated, by making the process of doing them more complex - at least when the underlying, and unseen by the user, programming level is included.

More correctly, of course, the *real* power of computing derives from harnessing the electrical states of the various electrical components to carry out tasks that are useful in human terms. We say, for example, that a computer does calculations. In fact, it does nothing of the kind, all that has happened is that the electrical potentials of some of its circuits have been altered in a systematic fashion. When we say that a computer has multipled two numbers we are speaking entirely in a metaphorical sense, it was the *humans* who set up the circuits of the computer that created the potential for the electrical circuits to *simulate* multiplication, and the computer user who actually *did* the multiplication. The circuits were used as an aid in performing the calculation instead of the more usual pencil and paper. Like a slide rule or its precursor, the abacus (or the pen and paper for that matter) a computer is just a tool, a means of carrying out tasks that exist in the real world, not in the electrical circuits of a machine or the beads of an abacus, and taken in its entirity - including the design and manufacture of the hardware and software - one would have to conclude that using a computer to do our sums is a massive complication overall (Harth 1993, p. 157), despite the fact that, from the point of view of the person doing the sums, it seems easier and more reliable, and a lot faster.

The point we are making is that a programming system exists only in the form of a metaphor. As with the calculation, the compilation process is a just matter of electrons moving around the circuits. Programming is just *thinking* about setting up a metaphor for some task in the real world in terms of electrical potentials, using another metaphor, the programming system itself - the human doesn't *do* the task, she sets up the electrical circuits to simulate it. But if most of the complexity in programming derives from what we cause our programming systems to do, then teaching programming is just a matter of addressing novices at the correct level in terms of the complexity of the material presented to them. As with life, the simple forms are coded first, and more complex forms build on what has gone before. However, this model of instruction is so blatantly obvious, that it could not be missed, and the real point is that it simply doesn't work. No matter how simple the code artefacts are kept at the introductory level, novices still struggle to learn how to program!

Clearly our analysis of the system in which programming takes place has failed to reveal the source of the difficulty of the learning experience, although, in fact, this could be argued two ways. If the *programming system* is seen as entirely embodied in the "programming language", then, yes, the system is the source of the difficulties. But our point is that it is a mistake to consider the language as *the programming system*, and that the difficulties stem from this mistake. Programming is *thinking*, designing a solution, not just coding, and the programming language is about coding not conceptual design. So it is not the programming system, taken as a two stage process, that is causing the difficulty, and if this is true then it has to be something else, and there are only two possibilities left, the 'learning system' and the 'teaching system', basically the psychology of the learner and the pedagogical environment.

Of course, the actual situation is likely to be caused by a mix of these two factors, as it is impossible to separate the teaching from the learning at any fundamental level of understanding. Moreover the 'programming system' does inevitably, as the artefact being studied, have some influence, but our point here is that it cannot be a major factor as it is not *inherently* complex within itself. Something about the way that we approach programming, both as a task to be done, and as a skill to be acquired, is causing it to appear difficult. This means that we are looking at two interfaces, that between the human mind and the domain of programming, and that between the programming system and the domain of education. Unfortunately perhaps, for an activity that has come to be fairly ubiquitous in modern societies, the history of programming is that it arose within the Computer Science domain, and it is the requirements of this domain that have dominated its progression over time.

One can say, 'unfortunately', here, for two reasons. Firstly, programming is *essentially* a *human* activity, and just as one does not look to engineers, in the form of bicycle manufacturers, "bicycle scientists", so to speak, for advice on the skill of riding bicycles, so one should not leave the development of *programming systems* (bike riding systems) entirely to people who are experts in the science of computers (the bike making system). And this points to the second reason, the creative, and therefore artistic, nature of programming. It is a recurring theme of our culture that, insofar as science is technical in nature, it does not cope well with human creativity. The requirements of any 'physical' system at a technical level are entirely 'logical' - a mechanical system runs according to the laws of mechanics, the 'logic' of mechanics if you like, an electrical one to electromagnetic principles, and so on. As Wittgenstein pointed out "logic is not a theory, but a reflection of the world" (Wittgenstein 1955, 6.13), it's about what follows "automatically" from what. Any abstract symbolic 'logic', therefore, is merely a representation of 'mechanical' action in a symbolic rather than a material domain. Calling a symbolic system 'logical' is making the claim that it functions *only* in accordance with a set of laws or rules just as a material mechanical system does. But any system, material or symbolic, that is driven by a set of laws, in other words, a logic, does not relate closely to the way that technical systems interact with human creativity.

Indeed Plato would have banished "the poets" from his "Republic" because he saw it as having a "technical" system of politics with a "technical" logic of its own, a true "political science", that would be disrupted by the participation of creative individuals in its political functioning. Any technical system that is meant to operate *mechanically* or at some level, automatically, by definition, does not require, or even maybe *easily accept*, human input - it is designed to be as automatic as possible. In a sense it has its own modus operandi, its own inherent *logic*. And this, indeed, is what is really meant by the term 'logic' - it is the means by which a mechanical or automatic operation proceeds, no matter what its status in terms of actual physical existence is. Such systems are not 'rational' in the sense of taking account of their environment, just 'logical', and that is precisely why they have to be 'programmed' or directed by a 'rational agent' if they are to result in activity that is 'rational', or meaningful in human terms. The very essence of Alexander's position is that any human process involves using a method that although it "is precise, ... cannot be used mechanically" (Alexander 1979, p. 12).

1.2 Rational versus Logical

The point at issue here is the 'intent', or lack of it, in a system. A mechanical system functions on the basis of a set of laws or rules only, in its own terms there is no 'intent' being expressed in its functioning. It simply functions according to the 'logic' of its mechanics. But this is not the way that human thinking functions because thinking arises from the requirements of a biological system in terms of survival. Biological being implies an intent, a purpose, that does not exist at the level of physical existence. There is no set of rules that can be expressed mechanically, no 'logic', that can guarantee survival as a biological being, this is a domain with a completely different set of forces that can only be dealt with contingently.

What is required by a system that 'intends' to survive is the capacity to react *rationally* to a situation, not logically in the sense of a fixed "automatic" response. Situations are *measured* against previous experience to guide reaction with the 'intent' of maximising survival potential, hence the *rational* status of action in such systems - the root of 'rational' is 'ratio' or 'measure'. This is why biological systems are *rational* rather than *logical* because no logic can account for context, at least not where context is essentially unpredictable. Any system which functions according to some 'intention' of its own must, perforce, be context-driven, not law-driven. The word 'intent' here does not necessarily imply any conscious purpose, just the necessity for a system to function in response to the current state of its immediate surroundings. That is, it functions in accordance with its 'intent', usually expressed as a set of needs of some kind, rather than functioning mechanically regardless of context.

The intent to survive is most clearly expressed as the making of decisions. This is so thoroughly embedded in the thinking process, probably deep down in the almost pre-conscious level (Luriia 1973) that Narciss Ach has called 'imageless knowing' (Association GREX n.d., p. 14), that we hardly realize that we are doing it, and even less do we notice how vital it is in terms of everyday living. Only when it is missing do we notice the dearth of creativity that results. There is a form of epilepsy that induces attacks of the kind known as 'petit mal' that result in human "automatism"[1], that is, the person undergoing an attack is unable to make decisions. Complex behaviour of the kind that does not require making decisions, for example, playing a musical instrument, continues during an attack, but the decision-making, and, interestingly, the memory[2], functions, are switched off (Penfield 1975, pp. 37-9). In a sense then, thinking, if indeed it can even be called by that name, has been reduced to the processing of pre-programmed behaviour, in effect, the execution of a program, and programs are purely syntactical (Searle 1990). There is no real meaning, no "intent", being expressed in purely automatic behaviour, what happens just happens because it is preprogrammed to happen.

We suggest that this means that the creative power of thinking lies in 'meaning' and that it derives from the ability, and the freedom, to make decisions. In fact Bertrand Russell has defined rationality in terms of making decisions based on available evidence (Russell 1961, p. 14). Most situations encompass a degree of uncertainty, so being rational requires subscribing to the most probable explanation based on the currently available evidence while keeping one's mind open to new evidence and new interpretations of old evidence (Rubinstein 1975), in other words, new meaning. Forced to work on purely programmed lines the brain becomes totally uncreative, and indeed, loses the facility of 'understanding'. In fact the 'mind' ceases to exist, the epileptic becomes "a mindless automaton" (Penfield 1975, p. 37). Moreover the lack of a memory trace (engram) during a 'petit mal' type epileptic attack corroborates that it is 'higher level' brain functioning related to being 'conscious', the 'mind' in other words, that is being affected by the attack (Penfield 1975, pp. 40-1). The pure neuronal logic being expressed by the pre-programmed behaviour, does not allow thinking in the normal 'conscious' human fashion to occur. So forcing the mind to use just computer logic is probably equivalent to forcing the brain to use just neuronal logic as in the petit mal situation, and it is most likely this restriction that causes programming to be so hard. Thinking involves non logical relationships as well as the logical ones. As Niels Bohr once said to a student, "you are not thinking, you are just being logical" (quoted in (Harth) 1993, p. xv).

The very point about a problem is that there is a degree of uncertainty in a situation - you don't *know* how to proceed, you are overwhelmed, in a sense, by the number of possibilities. So trying to make a strictly logical progression is not always the appropriate response, often you first have to 'discover' the premises or 'first principles' on which to base your logic. This means that you have to

[1]Note that this involuntary form of automatism is not to be confused with the automatism that drives expertise which is discussed at some length in Chapter 9, although it is interesting to speculate about how they might be related at the psychological level. That both the expert and the subject of a petit mal attack are, or can be in the case of the petit mal subject, involved in what appears to be complex behaviour, that is, exercising a high-level skill, cannot be entirely coincidental. Some fraction, at least, of skilled behaviour is programmed behaviour.

[2]This is another synergy with the "automatism" of skilled behaviour, given that skill automatism is based in memory (see Chapter 9).

apprehend the form or structure in the situation even if it is not immediately apparent, and it turns out that, even in the most seemingly chaotic and purely probabilistic systems, there are causal chains at work, which is saying little more, in actual fact, than that everything in the real world has a history, that nothing is free of context of some kind. This contextual "background" underlies the patterns we see in nature, and sets up the pattern languages that we use to make sense of the world, as much as it drives the causal chains of events.

> In the early 1960s, MIT researcher Edward Lorenz studied the weather's unpredictability. To his amazement as he was poring over his computer simulation, Lorenz discovered order amid the chaos. This seemingly disordered planet has remarkable patterns in its weather, geology, and chemistry. Computers, mathematical equations, and photographs have now verified these orderly forms. James Gleick's *Chaos: Making a New Science* tells a fascinating tale of this new field of study. (Fabian 1990, pp. 30-1)

The point is that there are patterns in nature, that is, order, even in seemingly disordered states. It is the major premise of this dissertation that solving almost any problem will involve dealing with the order inherent in a situation and that it is repeating form, pattern, that tells you about order.

In the activity of programming there is a duality involved that crystallizes around the two foci that we might call the 'science' and the 'art' of programming. If the calculus of probability has a place in programming it is in the 'space' between the 'science' and the 'art' of programming as a partial formalisation of a personal act. The 'science' is embodied in the machine logic and is completely formal, but the 'art' involves the personal expression of the formal rules, and the act of selecting among them, scientific hypothesising, is subject only to the laws of probability. The art of programming, then, is the skill in using the science, the logic, but it cannot be reduced to just those rules of logic, there are a set of 'rules of art' or maxims, as Polanyi calls them, that derive from probability (Polanyi 1958, pp. 49-50). These are not the same as the strict rules of formal logic expressed by the program, as "interpreting degrees of probability as degrees of truth is unintelligible" (Levi 2004, p. 461). Ultimately programming, like living, is about dealing with the probabilities inherent in a real situation, deciding what you believe to be the case, it is based on hypotheticals - partial beliefs, perhaps, not truth-values.

This, then, is the problem of *meaning* which we discuss at some length in Section 8.3. The very notion of 'concept' is built on 'intent' in this sense. So while concepts are, by their nature, abstractions, there is no sense in which this abstractedness expresses an essential disassociation from reality. While the initial association of an abstract token to a particular aspect of reality can be seen as arbitrary, the token can only function in terms of that association, that is, as a concept, as long as the association holds. Therefore, a concept is an expression of the *intent to associate* an abstract form with a real form, it is intended, if you like, to play a *causal* role in mental life. Any tiger-token is abstract only insofar that it is a mental artefact, not in its expression in terms of reality, its effect on behaviour. Any notion of creativity, therefore, exists because of the abstract nature of a system of tokens; in fact the tokens themselves are, or were, acts of creation. Tokens, on their own, don't mean anything, there was an act of creation in giving them meaning and and it was the 'intent' to bestow meaning that drove the act. Creativity is a particular form of the expression of intent.

As the human mental system is a product of evolution, it *must* reflect 'intention' in this sense. The very notion of 'symbol' rests on it "standing" for something in the real world, carrying a message, and, as Howard Pattee says about molecules, a sign "becomes a message only in the context of a larger system ... called a 'language"' (Pattee 1969, p. 8). A symbol is a "mental representation", and representation can be seen as nothing else but the expression of intention in the sense of carrying some meaning in terms of behaviour. As Fodor puts it, "'mental representations' are the primitive bearers of intentional content" (Fodor 1998, p. 7). In the end, a system that reflects reality only makes sense in the context of reality. Its *ultimate* "purpose" is to enable an organism to exist in the reality, nothing else, therefore it has to be contextual and it has to express intention because it drives the organism's behaviour in the real world, its organismic "Umwelt" in Jacob Von Uexküll's term. It is the powerhouse of rationality, of doing whatever is necessary in given circumstances. And here lies the conjunction with creativity, because a 'rational action' is, by definition, *created* to fit the context in a way that no logically decreed (programmed) action ever can be. Creativity is simply the result of the tension between intent and context.

The result of the development of programming having taken place within the confines of a technical domain, therefore, is that its nature as a *creative activity* has been, if not entirely overlooked, at least given only cursory attention. It's as if the activity of painting had been left to the "chemical scientists" who develop the paint, or literature to qualified grammarians. Realistically, the only input that should be expected of the experts who design a system that is to be used to express human creativity, is that they make the technical details as unobtrusive on the creative process, as invisible to the artist if you like, as possible. It is almost, if not entirely, a truism, that creative thinking has always to go outside of the established 'logic' of the domain in which it occurs. The really great breakthroughs, even in science itself, the big creative leaps, are made by people, like Copernicus, Galileo, Newton, Darwin, and Einstein, who are prepared to think outside of the strict confines, the 'logic', of their domain of interest, who, in fact, create the foundation for a new 'logic'. A point, often missed by scientists, and particularly, I must say, by computer scientists, is that the *practice* of any science is *always* an art.

So the trouble is that, as computer scientists, we have allowed the development of the *art* of programming to be driven by the way that the machine proceeds, its internal, or should that be, infernal, logic, rather than the way that human potential, at its most fundamental level, is expressed, *creative thinking.* If we are going to succeed better at teaching people to program we are going to have to see programming as less of a science than an art. A program expresses, most often, some need in terms of some human or 'real world' organisational system, not a need that exists in the computer system itself, the computer is just the means of executing the program. This is most clearly apparent when one comes to judge the "success" of a particular program, and here it is clear that a *successful* program is one that meets the needs of the system that commissioned it, not the system in which it happens to be represented. As perfectly as a program is designed in terms of the technical aspects of the computing system, it is worse than a complete failure, in terms of wasted resources, if it fails to do what the system for which it was designed required of it.

Moreover the confusion about what constitutes quality in a program surfaces in the nonsense notion of *elegance* in code. But it is interesting to note that it is impossible to define 'elegance' in terms of the execution of the code - a supposedly 'elegant' program will perform in the same manner as an equivalent 'inelegant' version, given, of course, that they are both 'correct' implementations of the specification. This suggests that 'elegance' is another informal rather than formal concept. When a program is said to be elegant what the adjective refers to is the *solution* expressed in the code, not the code itself. Elegance can live, surely, only at a conceptual level, code expresses concepts in terms of the operation of a computer, nothing more. So the key to programming is conceptual understanding, and the internal logic of the machine, on its own, can tell us very little about the creative process that produced it, nor yet, about its quality.

Experience in counseling novices makes it clear that there is only a tenuous relationship at best between the ability to 'understand' an already existing code artefact, and the ability to program. Moreover, this points to the use of examples in teaching material. The use of such examples is predicated on the basis that the code in the example will help novices develop conceptual understanding, and insofar as this is probably true in terms of the understanding required to *read* code, it is patently *not* true in terms of the understanding required to *write* code because the use of code examples in teaching material has been ubiquitous from day one of instruction in how to program.

The point here is that the code example is written in the *logic* of the machine, so the translation off the printed page is from logic form into concept form. But writing a program from scratch requires translation in the *opposite* direction and before one can even *translate* from concept form, the conceptual form has to be *created.* Hence the tenuous relationship between understanding written code and writing a program from scratch, the two processes are based on quite different 'understandings'. One is based on revealing the intent expressed by the programmer and the other on *creating* intent.

1.3 Logic and Creativity

Of course the troubled nature of the relationship between human creativity and the 'logics' of the various systems that we interact with has not gone unnoticed, and it is this dichotomy that has driven much of what we used to know as 'progress'. There is a sense in which much of history is a recounting of the steps involved in freeing human potential, that is, creativity, first from the immediate needs of survival, and then from the constraints implied by the systems that flowed from the previous step - in science this is expressed in the necessity for the periodic "scientific revolutions", identified by Thomas Kuhn (Kuhn 1962), where the previous paradigm (logic) is replaced by a new one.

The first great human tool, was communicative language, but, while it freed up the potential involved in co-operative action, it introduced the constraints involved in making meaning communicable. Insofar as language is an extension of thinking, it formalised the thinking process by introduction of the laws of discourse, the *logic* of communication if you like. Thus, in order to communicate, meanings have to be agreed - conventionally defined, in effect. But more than this, meanings larger than those that can be encompassed in a single symbol, at first auditory and later written, need a means of being expressed in a systematic and predictable way, so the various combinatorial systems - the rules for combining letters, spelling, the rules for combining single syllables, compounding, and the rules for combining words, grammar - evolve out of this need. In a sense, syntax, in all these forms, is the *logic* of a communicative system.

Fortunately, syntax is not *too* constraining of human creativity in thought, despite the best efforts of grammarians and pedants at linguistic engineering, because the language, as *the* system to which syntax is the logic, cannot be entirely externalised; it retains, of necessity, an internal and personal component. Nevertheless, the tension between language as dynamic meaning, and language as a means of communicating meaning, is a feature of our history. A feature that gives rise, no less, to the great human tradition that we know as literature - linguistic art - where expression breaks through the constraints to create meaning that did not exist before. There is an important sense in which meaning is always and entirely something that *belongs* to a particular individual, which is why it has to be conventionally defined in order to enable transmission, and, indeed, this notion of a sort of personal belonging gives rise, eventually, to the laws that attempt to confer ownership of sizable chunks of meaning on individuals, to protect, in effect, the personal component of meaning from the act of communication.

But even here there is an inevitable tension because no individual can communicate any idea except by means of the communicating system which, by definition *belongs* to, or can be credited to, no single individual. In a sense, no idea can be communicated except through the public domain, so the copyrighting of a particular collection of ideas violates the public nature of the act of communication. Thus what the copyright laws represent is the same essential misfit between the subjective and objective natures of the communicative act, between meaning and the *system* of communicating it. Language, as an enabler of the act of communication, is an attempt to bridge the gap between individual subjectivities, and the only way to do this is to, as much as possible, *objectify* it - that is make it external, just like a material object is, to both of the communicating subjective agents. Copyright law juggles the need of an individual to make a living out of one's thinking, with the essentially communal nature of so doing. And the interesting fact is that it accomplishes this paradoxical feat by reusing the laws of property, which apply to objects, and therefore it repeats and reinforces the objectification process involved in symbolising meaning.

But this objectification of meaning is a big problem, because meaning is a relationship between things, not a thing in itself. The meaning, *mythos*, has become the word, *logos*, and lost its essential place in the 'relatedness' of everything, its part in the *wholeness* of the world. If creativity is anything, it is the essential fluidity of relationship. By its nature, relationship is not fixed, indeed it may be that the very notion of time, almost our archetype of fluidity[3], is merely an expression of relationship between objects, the feature exposed by Einstein as Relativity. So what the *systemisation* of meaning to accommodate communication does is to restrict, to a large extent, the freedom, or fluidity, of meaning.

Any system of logic, by its very nature, rests on detailed analysis. If a domain is essentially predictable then one is saying that everything that occurs in it follows logically, or is deducible, from a set of premises. But this means that one needs to be sure that one's set of premises is complete, that one has *detailed* knowledge of the domain. The very force of industrialisation is the power of making a process predictable, indeed as automatic (un-manual?) as possible (Francois 1964, p. 30). But there is a commensurate issue of control. In order to make industrial systems safe, they have to be micro-managed at virtually every level of detail. This is why faults (bugs?) in large complex systems can be so hard to diagnose, they can be embedded at a level of detail that is not immediately apparent.

But the human system, like all biological systems, operates in ecological terms where goals are achieved by "making the most of robust, reliable sources of relevant order in the bodily or wordly environment" (Clark 2004) of the organism, not by micromanaging the system at every level of detail. This is not a world of precision and tolerances, of fine detail, one would would hardly ever inadvertently cause injury to oneself if this were true, but one of judgement (leaving open the possibility of misjudgement) and creative action. When one is *doing* something, washing up, for example, one is just doing it without any sense of controlling every aspect of muscular action involved in the task. The human, from infancy, has learned "by trial and error and practice, which neural commands bring about

[3]Most of the things that we refer to as 'flow' involve a change of some relationship between objects - *time* is, in effect our *measure* of the change in relationship between objects. This idea is elaborated in Section 5.4

which bodily effects" (Clark 2004) and has practiced them to the extent that they have become tools for the expression of creative activity. The essence of organism is behaviour. Organisms are opportunistic not logical systems, they *create* rather than *execute* behaviour.

If it is anything, a particular act of creation is, at its most fundamental level, a change in the *representation of reality* within some system that purports to order it, either physically or symbolically. Any building activity is changing the environment to better represent some need of human occupancy at a material level, while any building code is a way of representing the same needs symbolically, in terms of the goals, implicit and explicit, of the whole of society. By definition any act of creation cannot be *logical* in terms of the prior representation as it is a novel premise in terms of the logic of the representing system. It is a premise in the *new logic* that simply was not present in the old one, hence its creative status.

1.4 Programming as Rational Action

The canker at the very heart of Computer Science, then, the factor that makes programming, as an extension of the human facility of thinking, so difficult, is the nature of the machine. In terms of its actual functioning at the level of machine operation, a computer can be made to do nothing that is truly *novel*, the inherent 'logic' of its operation is fixed. But every program represents novelty in achieving action at some level of *human* organisation that was not possible previously. The creativeness exists in using the fixed logic of the machine to complete a task that was formerly done in some other fashion, or that was not done at all because it was too complex, time consuming, or difficult. In an important sense, the creative force in terms of programming is one that we are familiar with from language - metaphor. What a program *really* does is to express meaning in some domain of interest - if it is accounting software then it is the domain of finance, if it is typesetting then the domain is publishing, and so forth. But it expresses this meaning in terms of the larger system utilising the fixed operations of a machine that in themselves have no *meaning* in terms of the larger system.

Computing as a *science* is about the operation of the machine. As a *domain* it is about organising activity in virtually every field of human endeavour. In these other fields the actual details of the machine and its operation, the science of computing, are irrelevant, yet it is these operations that are used to express the meaning in the other fields. If it is anything, then, programming is the art of expressing meaning in systems that have *less than nothing* to do with computer science except to use the product of that science, computer systems, to drive productive purpose in their own terms. In other words a program is *pure metaphor.*

Where Computer Science can contribute, and indeed has contributed, is in making the creative use of the operations of the machine in these other fields as easy as possible, and this goal translates into the task of fitting the activity of programming to the way that humans think. This task has taken the form of developing ever higher levels of programming languages *where higher level means abstraction from the actual details of machine operation.* Moreover, in terms of easing the task of programming at a professional level, there is no doubt that these developments have been highly successful. It is not likely that computers would have achieved the scope of penetration into general human activity, nor the scale of the tasks automated that they have today, if programming had remained at machine language level. Nevertheless, developments in programming languages, basically the way that machine operations are represented to the programmer, seem not to have eased the task of teaching programming as much as would have been expected.

In other words, programming systems do not provide an easy mapping from real world thinking to machine operation, the algorithm that the average person would use in calculating, say, the average of a set of numbers does not translate easily into programming language form for a programming novice. Experts develop a programming model that takes account of " the mismatch between the subject's intuitive approach and the rather artificial constraints imposed by the behaviour" (Mulholland & Eisenstadt 2002, p. 1) of the programming system, but developing this is the reason for learning to program. This is therefore one of those classic "catch 22" situations that occur in education - how to help students leap across these "islands of understanding" (Mulholland & Eisenstadt 2002, p. 3).

> [We] found that although the fundamentals of iteration were not problematic to novices, the contorted mapping to specific language constructs was problematic: novices seemed to prefer to apply nested functions to aggregate data objects rather than cycling through individually-indexed objects. All of these studies confirmed our view that novices employed quite sensible models of the world, but that programming language instructors in general (including ourselves) consistently failed in helping novices to map their pre-existing models onto the specific ones required to deal with programming.
>
> (Mulholland & Eisenstadt 2002, p. 3)

As with other domains, notably mathematics, where the representing notation is formal and strictly logical in operation, learning to use them is seen as difficult. It is said that curriculum is essentially that which is taught in schools, both intentionally and unintentionally (Groundwater-Smith et al 2001). But it follows from this extended notion that the certainly unintended outcome that most students come to believe that programming is difficult is an aspect of the programming curriculum, the "hidden curriculum" in Groundwater-Smith's sense. Moreover, representing the intended component as the 'official curriculum', and the unintended effects as the 'hidden curriculum', is thus a way of counterposing the 'expected' results of instruction with what actually happens. In effect, the 'hidden' and 'official' curricula are in a dynamic inverse relationship to each other and the relationship is telling us something significant about our pedagogy. The fact that many universities feel that they "have to 'hide' the maths components of courses, in case it puts potential students off" (Sheffield Hallam University 2002) demonstrates the pedagogical significance of this relationship as most students of mathematics similarly come to believe, *learn*, that maths is difficult.

The hidden curriculum, as "an experiential rather than physical phenomenon" (Arizona State University 2000) is thus, to some extent, an expression of the 'psychological field' of students. Goals, expressed in and by the official curriculum, are modified during the learning experience (hidden curriculum) to become the actual outcomes achieved by means of the 'psychological field'[4] of the students. This is an important idea in educational thinking because it converts the notion of the curriculum as a whole from a static document conveying information about what is to be taught in the classroom to a dynamic element in the overall educational process. It explicates the students' state of readiness, a product of their 'psychological field', to absorb the teaching material and methods that have been applied to them. In effect, it 'measures' the effectiveness of the curriculum as a whole, and in so doing presents us with a view of the students' status as learners.

The fact that both of these areas, programming and mathematics, involve using a formal system of logic to express meaning, that is, to create new meaning, and that both are seen to be difficult to use, and more particularly, difficult to learn, cannot be entirely coincidental. From this it appears that the job of the official curriculum is to attempt to counter the belief that these activities are difficult that derives from their nature as logical systems. It would seem that the effort to disguise the strict logic of the machine behind programming systems

[4]The ability to acquire a skill like programming relates more to the possession of a complex nervous system than to any particular features of it. Any event or process in a person's life is guided by a multitude of factors that form a sort of psychological field, where a 'field' is a totality of coexisting mutually interdepedant elements, in which her life takes place. This 'life space' is the psychological environment and is what we mean when we refer to a persons 'psyche', that complex web of needs, desires, motivation, mood, goals, anxiety, ideals, and so forth. The topology of this space is the result of personal history and determines how the person will react to new experiences. "Any behaviour or any other change in a psychological field depends only upon the psychological field *at that time*" [emphasis in original] (Lewin 1951, p. 44-5). In understanding human behaviour this 'psychological field' is more important than the more physically or materially 'real' biological and physiological factors. This implies that social setting is just as 'real' as the empirical 'physical' factors of a person's environment. From the first day of life we are objectively part of a social 'mileu' and would die in quick order if we were to be removed from it. So the 'subjective' psychological world of the individual, the 'psychological field', is influenced by social relationships right from the start. This is why children are extremely good at picking up fine nuances of mood, and in particular of emotion in adult speech. Preverbal children can discern and discriminate between sounds expressing approval, disapproval, happiness, and anger ((Fernald 1993) (Haviland & Lelwica 1987)) in their environment from a very early age, and are capable of producing sounds with these characteristics (Joseph 1982).

that are closer to natural languages has failed in curriculum terms, even though it seems to have been highly successful at the professional or industry level. This tells us that, as a metaphor for representing the relationship between thinking and the operations of a machine, 'language' has taken us as far as it is going to in making programming easier to learn.

It is interesting, however, that the recent transition from procedural to object-oriented programming languages was, at the very least, an attempt to take a partial step away from the 'language' metaphor. The idea that the task of programming can be seen in terms of objects, characteristics of objects, and relationships between them, is saying more than "programming is communicating", it is saying that "programming is thinking" because the object-oriented view of the world is simple *metaphysics*. That is, it is an attempt to implement the way that we relate to reality rather than the way that we communicate, it is based on the sort of thinking one would do in solving a problem in the 'real world' rather than that involved in communicating.

One often hears it said that Simula was the first object-oriented programming language, but this is, as a matter of historical fact, an oversimplification. It is true that Simula introduced the concept of *class* but the entire language was not built around the *class* concept in the way that Smalltalk is, and, strictly speaking, Simula is thereby an *ancestor* of object-oriented programming language not itself one (Horowitz 1983, p. 409). What is significant about this is the conjunction of the object-based nature of Smalltalk with its pedagogical intent, and, in this respect, it is interesting how the developers of Smalltalk discuss what it is that they thought they were doing in designing a programming system for children. The first line of Alan Kay's abstract of a paper outlining the early history of Smalltalk actually states that they saw themselves as applying a *pattern* from the real world - the pattern of biological organisation.

> Most ideas come from previous ideas. The sixties, particularly in the ARPA community, gave rise to a host of notions about "human-computer symbiosis" through interactive time-shared computers, graphics screens and pointing devices. Advanced computer languages were invented to simulate complex systems such as oil refineries and semi-intelligent behavior. The soon-to-follow paradigm shift of modern personal computing, overlapping window interfaces, and object-oriented design came from seeing the work of the sixties as something more than a "better old thing". This is, more than a better way: to do mainframe computing; for end-users to invoke functionality; to make data structures more abstract. Instead the promise of exponential growth in computing /$/ volume demanded that the sixties be regarded as "almost a new thing" and to find out what the actual "new things" might be. For example, one would compute with a handheld "Dynabook" in a way that would not be possible on a shared mainframe; millions of potential users meant that the user interface would

> have to become a learning environment along the lines of Montessori and Bruner; and needs for large scope, reduction in complexity, and end-user literacy would require that data and control structures be done away with in favor of a more *biological scheme of protected universal cells interacting only through messages* [emphasis added] that could mimic any desired behavior.
> Early Smalltalk was the first complete realization of these new points of view as parented by its many predecessors in hardware, language and user interface design. It became the exemplar of the new computing, in part, because we were actually trying for a qualitative shift in belief structures - a new Kuhnian paradigm in the same spirit as the invention of the printing press - and thus took highly extreme positions which almost forced these new styles to be invented.
>
> (Kay 1993, p. 69)

This is *almost* saying that a programming system needs to be more than a compiler. Its pedagogical flavour is only to be expected in a system designed explicitly for children, but the fact that there are are major Platonic overtones as well, as Kay goes on to elaborate, comes as something of a surprise.

> Smalltalk's design - and existence - is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages. Philosophically, Smalltalk's objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealisations of concepts - Ideas - from which manifestations can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of Manifestation-Idea - which is a-kind-of itself, so that the system is completely self-describing - would have been appreciated by Plato as an extremely practical joke.
>
> (Kay 1993, p. 70)

Unfortunately, implementing, in a compiler, the way that we see the world is almost certainly impossible, just as our attempts to implement natural languages in compiler form have proven difficult. Moreover it is probably true to say that getting close to natural forms of linguistic expression in a compiler interface is likely to be a lot easier than getting close to expressing a metaphysic. We hardly do that well even in natural language! In the end a computer operates *according to the logic of the computer system*, not according to its relationship to the world as a *rational agent* does.

Nevertheless the underlying idea, that "programming is thinking", has to be fundamentally correct. As teachers of programming we are closer to being educators in Arts rather than Science. After all, if the attempt to teach the higher level programming languages made us more like natural language teachers than science teachers, then the attempt to separate the thinking from the coding should make us more like teachers of philosophy than teachers of (software) engineering. For that is what the introduction of the pattern language idea into computing really represents, an attempt to separate programming into two distinct activities - thinking about the problem and its solution in conceptual terms, *design*, and coding the conceptual solution in the logic system of the machine, *implementation*. In teaching programming we are essentially teaching *humans* to solve problems that exist in the real world with a particular tool, not to understand how the tool works. This makes us philosophers, not scientists or engineers, nor even logicians - we are teaching people how to *think*. The job of solving a problem in the real world is the job of a *rational agent* not a computer scientist, we are generalists not specialists.

But, if anything, this generalist nature of programming, problem solving, is a huge advantage because the essential nature of human beings is itself generalist in flavour. Our lifestyle is based on solving problems, on adapting to novel situations, on expressing creativity. Given the correct approach, an appeal to our native "intelligence", the problem of teaching people to program is not insurmountable. Of course, this does not mean that it will ever be easy, but the *real* problem is not the teaching, but finding the optimum way to teach, the right way to address the mind. So the core problem is identifying those features that make us, as a species, successful generalists, experts at solving problems, creative in spirit, and, here, there is only one possible candidate - the way that we think.

From the difficulties we see in the current pedagogy it is clear that humans do not think logically, we are not programmed automata, we are organic beings living in a real world. The striking thing about life is that it is full of patterns in the sense that patterns are collections of *generalised* features. By this we mean that, given a particular concept to think about, say "tree", one is very unlikely to think of a particular example of the concept, a *particular* tree; rather the mental "image" will be short on particular features and strong on those features that are general, common to all the objects we would classify as "trees" - features like trunk, branches, leaves, bark and roots. This is hardly surprising, because a particular tree is rigidly defined by its features just as a computer program is rigidly defined by its code. In other words, an actual tree is, in fact, a sort of a program, specified genetically. Reproducing it abstractly, in the brain, is quite beyond our means. Even when prompted to think of a particular tree, we will still only be able to represent it in the most general way - the tree with the broken branch, the one near the lake, and so on.

Patterns are important as the basis for concepts because they free the mind from any detail that is irrelevant to our current purpose, that does not figure as significant in the immediate context. Because our concept for 'tree' has to encompass entities as diverse as palm trees and Araucarias it needs to be generalised. Indeed, for most purposes, it does not even matter that we are able to correctly, or should that be, conventionally, define every type of tree under our concept. And this point relates to what is a fair question arising from the first sentence in this paragraph: aren't then patterns the same thing as concepts? Patterns are not the same thing as concepts because the concept that a person has about a particular aspect of experience will depend on the needs of that person in terms of relating to that aspect of the world.

So, to revert to our 'tree' example, for most people the concept of 'tree' will be based on few patterns, maybe even just a single pattern. This is because they do not require the ability to use the concept in a precise way. The ability to identify an object as being a tree, rather than a shrub, say, to any degree of precision, is not an important factor in their ordinary lives. However, if there is some purpose for them in being able to identify trees to some level of botanical classification, then their concept of tree will need to be capable of dynamic refinement, that is, it will need, in fact, to consist of a pattern language. For that is the very purpose of a pattern language - to empower the process of *refinement* of a concept to greater and greater degrees of precision, in terms of the task at hand. That is, "there are levels [of abstraction] which are more significant in the problem space, and useful to the solution design than others" (Fincher 1999*b*). And, one might add, useful at *different* stages in the design process.

In pattern language terms we say that each pattern in a language *adds structure* to the pattern above it in the language and thereby *refines* our ability to use the concept for the purpose for which we need it by 'adding structure' to it. This refinement aspect of pattern language is the subject of Chapter 7, where the concept that is being refined, the pattern to which structure is being added is 'program'. In writing a program to do a particular job one is simply, at a conceptual level, redefining the 'program' concept to encompass the particular features required by the program that does the job. So the reason that a particular person needs a more structured understanding, a pattern language rather than a single pattern, of the concept 'program', which, after all, is a concept that virtually everybody in a modern society has in some form, is that the person is learning to write programs, not just to use them[5].

Our use of concepts is so fundamental that we usually overlook the fact that they were 'created' out of some situation where current knowledge was no longer adequate to explain it. So, for example, we commonly use the concept 'dinosaur', and can hardly envision not having it (Colbert 1971, p. 34). But there was a time when no such idea existed - the concept of 'dinosaur' *had to be invented* (Colbert 1971, p. 43). The significance of this 'creativity' at the very basis of

[5]So, for a casual 'user' of computers the concept 'program' probably consists of a simple tree, with arcs leading from the root node, 'program', to a few common applications, 'spreadsheet', 'word processor', 'web browser', 'calendar', 'calculator', and so forth. Most such casual users probably have only the haziest idea that the operating system that they interface with is itself a series of programs, let alone that a programming language is, in fact, a program that translates the programming language into machine code and that using one involves a dynamic refinement of the 'program' concept.

thought cannot be overstated. It demonstrates that 'creativity' is a function of thought, not simply an attribute of the individual thinker. We were, in effect, 'forced' by the situation to create the concept by the new patterns of experience arising from the discovery of extremely large fossilised bones and footprints that quickly outgrew the explanations of the bones of giant humans (Colbert 1971, p. 17) and the footprints of "Noah's raven" initially given to them (Colbert 1971, p. 52). If the 'stuff' of which our thinking process is made is itself a product of a *problem arising out of discovery* (a pattern) like this then it is impossible to explain the human condition in terms other than pattern language. Without the means to assemble the facts of experience into conceptual form, nothing of the human condition would be possible.

So perhaps the most significant thing to say about patterns is that they are so much a part of the very *way* that we think that we are genuinely unaware of their presence. Who is aware, to continue with the 'tree' example, that the ability to distinguish one type of tree from another involves the use of a pattern language? Indeed we often react with surprise when patterns are pointed out to us. For example, the approximately spherical shape of stars is just something that we assume, even though, in fact, we have no direct evidence that any star, apart from our own Sun, is actually a spherical approximate. This is quite striking when you think about it. Here we are, given one solitary example, populating a whole universe with "spherical" star objects when we cannot even resolve them in our observing instruments as anything other than points of light in the darkness. Why does no-one expect to eventually discover hemispherical, rectangular, conical, or even single point stars elsewhere?

The point is that "sphere" is simply a pattern in our experience of three dimensional fluidity. Freed, even partially and temporarily, from the effects of the gravity of our planet and of contact with solid surface, as in a free-falling drop of water, a quantity of fluid material will resolve the various forces acting on it by forming a number of "spheres"[6]. Even before we knew of all these forces in any detail, molecular cohesion, surface tension, gravity, exclusion and the like, *sphere* was simply a pattern of our experience of fluid substance. This *pattern* even came to explain the "spherical" shape of the planets, including our own, as we discovered the planetary origin in gaseous and liquid form. So "sphere" is simply our *pattern* for realising "natural" form in substance that is, or has, at some stage, been in, a fluid state in a 'neutral' gravity context. The discovery of a cube-shaped object in another stellar system would scream "consciously manufactured" at us! It does not fit our pattern for 'planet'. Rather it fits our 'artefact' pattern.

Considered carefully all knowledge is structural in nature. A sequence of events in any causal sense is purely linear in the temporal dimension, but a straight sequence of cause-event pairs, each event acting as the cause of a subse-

[6]Another example of this pattern is "bubble", where a fluid, usually a gas in this case, is "contained" by a membrane of some kind.

quent event and so on, is rarely, if ever, indeed, sufficient in terms of explanation. There are effects other than the direct cause that impact on the unfolding of each event and these arise from the connections between entities in the world, that is the *structure* of the system as a whole. Therefore, there is no understanding of the world, no knowledge, that is anything other than structural in nature. Even a sequence of bits derives its 'information' content through the relationships between subsets that are not apparent in a simple linear reading of the sequence - it is *patterns* in the sequence that carry any 'information', not the bits themselves.

The essential difference between the way that a computer works and the way that the human mind works lies in the means by which sequence is treated. To a human, experience, that is, a sequence of impressions, is 'read', processed, if you like, at the pattern level rather than at the level of the individual events that contribute to the the impressions. Most real world situations are simply too complex, there are too many things happening either simultaneously, or so close to it as to be effectively simultaneous, for us to be able to function as strictly sequential processors. Meaning is not how we *interpret* reality, it is how we *experience* it, the patterns are fundamental, not the strict sequence of objective 'events' that constitutes experience. Given the need to reconstruct the strict temporal order of an experience, we struggle - the experience, more often than otherwise, makes sense because of the various 'meanings' it carries, its 'structural' order, not its sequential order.

Therefore the task of programming a computer is to set up structural relationships in a device that processes sequentially. That is, unlike the human, the patterns are not 'read' directly, the individual bits are 'read' as a simple sequence and then stored as subsets, and it is from the collection of these subset units, that patterns meaningful to humans are derived. That is, meaning and structure arise from a direct 'reading' of the sequence by the machine, this is an entirely mechanical and logical process. The job of programmer is therefore to put the patterns into the sequence, to 'order' the mechanical functioning of the machine so that it fulfills a task that is *meaningful* to the user. This is the equivalent of having to deal with real life experience at the individual event level, it is just not what we *do* as functioning human beings. Our whole existence revolves around the experiencing and comprehending of meaningful patterns, of structuring reality in our mind as it happens, not just experiencing it raw.

So pattern is the absolute basis of our thinking. Before we can even work out the details of the various systems that *cause* the patterns we see in life, the logic, if you like, behind them - the forces that are "resolved", and the rules and laws that drive the "resolution" - we can detect and use them in structuring our lives. We detected, for example, *metal*, as a pattern of material substance long before we knew anything about the details of atomic structure, and, indeed, for most of us, our everyday experience of metals occurs at this level to this day. Our thinking is built on pattern, not detail. One doesn't have to know *why* metals are mostly 'cold', 'hard', and 'unyielding', and even knowledge of the exceptions to that ‘rule’ do not throw our mental picture. For that is the strength of a pattern. It doesn’t have to be *logically* consistent, or even definable in complete detail, it just has to work in helping us cope with the contingencies of life, in *patterning* our behaviour, in structuring our thinking. Patterns are how we think! It is patterns that give concepts the dynamic nature that static symbolic form struggles to keep abreast of, and it is patterns which drive the creative aspects of thought.

1.5 The Pattern of this Thesis

The pattern for the rest of this thesis is an examination of pattern language in the context of learning to program. This examination is attempted mainly at a theoretical level because it is not clear that the potential benefits of the pattern language idea can be fully realised until the full import of Christopher Alexander’s ideas are assimilated, as he himself has pointed out (Alexander 1999). We have avoided, as much as possible, giving examples of particular patterns, and where we have presented pattern languages we have kept them as general and as simple as possible. Our intention is to expose the pattern process as much as possible, and the particular examples that we use are directed to this end. So, for example, the material used in the step-through of the pattern process in Chapter 7 is not meant to be construed as an exemplar of pedagogical presentation, it is merely there to drive the process of deriving the solution. In any realistic pedagogical context, the patterns, pattern languages, and example problems presented to students would need to be developed to that context. This, after all, is the main argument that we present here, that, in the end, everything is context - the creative power of mind is relationship, not detail.

It is in this sense that the difficulties that people encounter in leaning to program are precisely an expression of the mismatch between the way that normal human thinking proceeds and the logical rigor of computer programming. If this impasse is to be avoided then it will be done by bringing programming closer to normal human thinking patterns. Chapter 2 is an overview of prior work that relates to the use of pattern languages in learning to program. The third chapter attempts an examination of the roots of creativity in experience, and Chapter 4 proposes that ignoring the basic fact that the practice of any science is, in fact, an art, is the source of the difficulties in teaching all scientific disciplines. Generating novel insight is the font of all creativity, and Chapter 5 argues that language is the means of doing this, and that, therefore, language is fundamentally about understanding, *not* communication. This is why pattern language is so vital in informing any creative activity, and Chapter 6 examines this connection. Chapter 7 works through a simple programming problem to demonstrate how a pattern language drives the process of building the solution at the conceptual level, and how it contributes to programming pedagogy in the wider sense. In the early chapters, the notion of the “epistemic disjunction” between subject and object was introduced, and Chapter 8 relates how the problem of meaning impacts on the psychology of symbolic behaviours like programming. Crossing the epistemic gap, creating meaning, is the fundamental psychological issue in terms of learning to program and Chapter 9 examines the use of 'semantic memory' in addressing the cognitive overload caused by many creative activities. Although the main thrust of this project has been in developing a theoretical basis for the use of pattern languages in programming instruction, we did attempt three trials of the theory, and these are discussed in Chapter 10.

So the essence of our argument is this. Life is experienced - even perception itself is finally an artefact, constructed from, not just a copy of, 'reality' (Piaget 1972, p. 27). After all, "reality and world ... are just titles for certain valid unities of meaning" (Edmund Husserl quoted in (Crosson) 1967, p. 127). It is out of experience that the human property of *meaning* arises through the apprehension of the recurring features we perceive, the common *patterns of life*.

> [No one can] deal in the unique fact, because facts and events require reference to common experience, to conventional frameworks, to (in short) the general before they acquire meaning. The unique event is a freak and a frustration: if it is really unique - can never recur in meaning or in implication - it lacks every measurable dimension and cannot be assessed. ... Facts and events (and people) must be individual and particular: *like* other entities of a similar kind, but never entirely identical with them. That is to say, they are to be treated as peculiar to themselves and not as indistinguishable statistical units or elements in an equation; but they are linked and rendered comprehensible by kinship, by common possessions, by universal qualities present in differing proportions and arrangements.
>
> (Elton 1969, p. 23)

It is from this fundamental aspect of life, the meaning that is derived from relationship, from patterns of experience - *pattern language* - that the communicative form of language develops. but in order to make this transition *meaning* has to be artificially, and therefore conventionally, tied to signs. In a sense the human species has *created* a new world, a world of symbols, a *virtual* world in which life is not *experienced* but *symbolised*. So there is a true gap here between the *living form*, represented by *mind* and *imagination*, and the *symbolic form* of sign and syntax - Howard Pattee calls this split the "epistemic cut" (Pattee 2001*b*). Coding uses the second of these, *symbolic form*, and it therefore resides in the second *virtual* world, not the real world of life, there are no *meanings*, acquired from experience, for the novice here. Ultimately there is only one way to acquire *meaning*, and that is through experience, and only one form in which it can be directly represented, *pattern language*.

Of course, the epistemic split between syntax and semantics, symbol and referent, observer and observed, real and virtual is essential, it is the source of the power of abstraction - without it symbolisation is vacuous, "abstractions do not designate phenomena at all, but serve to describe them" (Langer 1962, p. 5).

> When physicists (or other scientists) use mathematics to describe events, they tacitly assume that the formal syntactic manipulation of the mathematical symbols they are using are not in any way causally influenced or restricted by physical laws ('Semiotic interactions do not take the place of physical necessity'). On the other hand, when measuring the initial conditions for a particular system, the symbolic results (the semantics) of the measurement must be assumed to be directly caused by the physical state of the system and not in any way influenced by the formal syntax of the mathematics.
>
> (Pattee 2001*a*, p. 350)

But this epistemic irreducibility leaves us with the real difficulty of what might be called the "rational relation" between symbolic and real form, the problem of 'meaning'. Indeed, it is perhaps, the cost of the process of abstraction, the cognitive overhead involved in establishing and maintaining meaning (Blackwell et al 2002) that lies behind the pedagogical problem in programming as we discuss further in Chapter 8.

The novice struggles with the symbolic form, programming language, because, to the person with no experience, the symbols are mere signs, they carry no meaning. The operations of the symbolic system are, in effect, reduced to syntactical combination, but there is no scope for creativity in non-semantic operations. What the process of learning to program is, then, is the struggle to acquire *meaning* from experience, to develop *pattern language* from using the purely symbolic form. Learning to program is a struggle because this is a reversal of the usual human progression, from *pattern language*, meaning, to symbolic system, communicative form - we understand before we communicate. Many of the negative phenomena that we observe in the programming situation derive from this reversal. Firstly, programming bugs are a manifestation of trying to communicate with the compiler on the basis of a 'misunderstanding'. Secondly, the failure of many programming projects is caused by the failure to 'understand' the users' requirements *before* the coding process starts. And, finally and most pertinently, expert programmers rely on knowledge, understanding, that they find difficult to transmit to others, precisely because they have acquired the *pattern language* of programming without ever using it consciously, or even being aware that they have it.

In other words the expert has constructed "meaning" out of experience, the only way that "meaning" can be acquired, to be sure, but she has done it through learning to *communicate with a compiler* rather than by learning to understand directly. Learning to understand first has to be easier, and this means that learning should be based on *pattern language* not programming language. All true understanding is conceptual in nature, so the novice should be presented with the required knowledge in conceptual form, not just its symbolic form. What we combine when we create are meanings, not symbols, the symbolic form merely represents the new meaning that has been created, it is post facto, and, as Dijkstra points out this often leads to the need to invent a new language to talk about a new concept (Dijkstra 1982, p. 342). And just as it is difficult to create new meaning purely by manipulating signs, so it is with the creation of new understanding. Learning is the creation in the learner of new understanding, so, as programming without understanding is difficult, so is learning anything without understanding, in fact, if anything, doubly so in the case of learning programming. The creation of *meaning* out of experience, that is, *understanding*, is the very essence of the human condition, and, as we attempt to show in Chapter 3, it is driven by *pattern language*.

So, in many respects this dissertation is a plea for a theory of programming. We believe that such a theory is required in order to provide a firm foundation for the development of a comprehensive philosophy around the activity of computing. Any attempt to develop a theory of programming must begin with the pedagogy of programming because it is in this context that the difficulties that programming causes the human mind are thrown into sharpest relief. The development of the modern computer has given us an unparalleled opportunity to explore the interface between mind and logic but the opportunity is currently being missed by the insistence that programming remain a component of Computer Science.

Programming is a fundamentally creative activity, it takes place in the mind, not in a computer. The manifestation of the activity that concerns the machine, the execution of the program, is merely the end result of the activity, and what the difficulties thrown up by programming tell us is that "it is not only the result which is important, but the process too. Not only the form of the results, but the form of the path that led to them" (Alexander 1964, p. 133). It is clear that the path, the *activity* of programming, has very little to do with the science of computers because the problems that it causes all occur in the mind and in the classroom.

So, although everyone involved in education agrees that "much of what we do as educators is devoted to conveying to the student the cognitive maps that we use for problem solving in a discipline" (Hiltz & Turoff 2005, p. 62) this is simply not reflected in everyday teaching practice. One can search high and low through most of the material used in instruction for the presence of any such structured representations of the basic knowledge involved and rarely find it. This means that the conveyance "to the student of the cognitive map" is not being done explicitly and the educator, therefore, must be relying on some principle of 'osmosis'.

The implication is that from the experience of learning to code students "will learn complex, transferable skills (analysis, design, problemsolving)" (Fincher 1999*a*, p. 5) rather than regarding "the acquisition of [these] skills as the ultimate objective" (Fincher 1999*a*, p. 5) of instruction. Somehow the correct representation for solving problems, the "cognitive map", is developing, "magically", in the students' heads from the linear presentation of largely unstructured knowledge that we use in instruction. We happen to believe that this is not only unlikely, but that it is the lack of such explicit representations of the structure of knowledge that makes the creative application of knowledge nigh on impossible for *most* students. The almost casual acceptance of the 'osmosis theory' of skill acquisition is one of the main arguments for a philosophical examination of the bases of programming education on the basis that "what *seems* commonsensical and unquestionable, and cannot for practical reasons be seriously doubted, may nevertheless be worth doubting" (Gellner 1968, p. 99).

Of course the appropriate cognitive map *will* develop over time, but this is a slow and laborious process. The trouble is that, ultimately, any view of the way that humans function in the world depends on memory. Everything that happens occurs in the present, so nothing that happens to us can project into the future *except* through the agency of memory. Without such storage of some kind of a *representation* of experience no *learning* of any kind is possible. Therefore we depend entirely on the way that experience is translated into useful knowledge through the use of memory. It would seem to be a given that, initially, experience is handled linearly, that is, stored as "episodic memory" in Tulving's sense (Tulving & Thompson 1973). The trouble with episodic or linear storage of experience is that, as a driver of future behaviour it can be seen to be capable only of reproducing the same sequence of events. That is, it would be entirely *mechanical* in functioning because the form of an episode is a sequence of events, one following, apparently logically, from the previous event.

But we know that this is not how humans behave, so the episodic memory which is *useful* only in terms of a mechanical or almost automatic response to novel situations *must* be transformed into a form useful in terms of the type of behaviour that we observe in humans. There is really only one way that we can envisage this transformation occurring, and that is the *structure* of the memory must change from the straight linear representation of experience to some other form that makes it more general and thereby more flexible. In other words we *create* meaning out of experience, and the change in the structure of the knowledge in memory must reflect this, must make the brain a processor of *meanings* rather than a processor of raw memory events. Episodic memory is converted into *semantic memory*, a concept that is explored in Chapter 9, and this is a change in *form*, a *structural* change that makes memory non-specific, that is generic. These so-called "Generic Knowledge Structures" (Graesser & Clark 1985, p. 32-3) are the cognitive maps of problem solving behaviour. All the generalising power of human thought comes out of this semantic structure. I can never know from direct experience, for example, that the Moon has another side, but I *can* infer it from my accumulated general experience of spherical objects. It is this generalising factor that converts memory into knowledge, that *makes memory useful.* Without it memory would be little more than a mechanism for replicating past experience exactly.

And, as we have seen, there is almost no explicit acknowledgement in pedagogical material of this essential structuring of knowledge. Students are left to their own devices in developing it. There just has to be a better way of assisting the development of cognitive structure, and such a way is suggested by the pattern language concept. Left to their own devices novices will concentrate on code-level information in lieu of the higher level conceptual structure that experts possess (Davies 1993). But this is a qualitative difference that will be difficult to bridge as is demonstrated by the fact that there is a stage in the development of expertise where the number of errors made temporarily increases as the restructuring of the knowledge base occurs (Lesgold et al 1988) (Davies 1994). Therefore providing novices with a representation of the knowledge structure, an evolving pattern language, right from the start should be a way of minimising this qualitative difference.

In fact, if one examines the very idea expressed in the phrase "cognitive maps that we use for problem solving" closely, it is apparent that something important is being indicated here. The purpose of a map is to give direction, to find one's way around, in this case, in a domain where a problem exists. But go back one step in the construction of this metaphor, that is, to the idea of a map of physical terrain, and consider that mental versions of maps of physical locations ('mental physical locality maps') take a great deal of effort and time to build. Moreover, even given that sufficient time has been invested in one's mental image of one's normal 'haunts', it is clear that such models are limited in the detail and reliability of their representation of physical space. As soon as any fine detail or non-routine travel is required we invariably resort to an external, that is, a non-mental map.

But even more telling is what we do when we are asked, off the cuff, to give directions to some stranger to our district. Initially we attempt to give our advice verbally, but if the instructions are at all complicated we either end up confusing ourselves to the extent that we realize that we are not helping our lost visitors or realise from the start that this is the likely outcome of what we are attempting. So we respond by resort to an external representation of the locality, a physical map - if necessary one drawn from memory, the proverbial 'mud map'. In other words, our experience of the very source of the 'cognitive map' metaphor tells us two things - that maps inform a process, and that maps are *best communicated* in written, that is *external*, visible, form. It is clear that when you are *instructing* some other person's travel (a process) through territory that is unfamiliar to them an external representation of your own knowledge is virtually indispensable. And if this is so for traveling in physical space then it should be obvious that providing directions for *any* process through *any* conceptual domain similarly *requires the map to be drawn.*

Yet, in programming instruction we *never* draw the map. By its very nature, programming compounds this issue because the knowledge involved in instruction is entirely pointless apart from its use in the activity of programming, one only learns this stuff in order to write programs, it carries no other possible benefit as knowledge in its own right. So programming is particularly difficult because it depends *always* on that vital ability to apply, rather than to just have, knowledge. In the end, the difficulty of programming is likely to be illuminated more thoroughly by philosophical rather than empirical methodology because there is little in the way of *measurable* factors involved.

Even the questions about the execution of a program come down, ultimately, to a subjective judgement. So, for example, measuring the quality of a program devolves to the question “does it meet the specification of requirements?” But this, in turn, just raises the question about *whose* interpretation of the requirements is to be used as the standard for measurement. Most of the so-called “crisis” in software engineering turns exactly on this question, because the problem can only be expressed in informal terms, but its solution will be expressed in formal terms - that is, precisely defined. The point is that the translation from the actual real world problem in the domain of interest to its solution in the form of a computer program is the creation of a metaphor, a journey from the actual to the ideal, the concrete to the abstract, the writing of the actual code is merely the final stage translation from the subjective understanding to an objective realisation. Like all journeys this is better understood as a process of becoming rather than a state of being, and as with all journeys one needs a map showing the relationship between the topographical features, not just the context-free knowledge that they exist.

Chapter 2

Prior Art

Out of intense complexities, intense simplicities emerge.

Winston Churchill

Concept maps are visual graphs, consisting of nodes, which represent concepts, and arcs, which represent relationships between the concepts. Concept maps are used in a wide variety of disciplines because of their ability to make complex information structures explicit. There are a wide variety of concept mapping languages ranging from informal to formal; all share the same fundamental structure (nodes and arcs), but they vary in many ways, including degree of formality (typing), allowable component types, component graphical attributes, inclusion of contexts, and miscellaneous constraints.

(Kremer 1997, p. iii)

2.1 The Philosophy of this Project

The idea of considering the potential for the use of the software design pattern concept in introductory programming courses necessarily involves consideration of issues in three broad areas, computers and computing, education, and psychology. Therefore research has ranged widely over issues in these three disciplines. But, more significantly perhaps, it is the nature of a project that encompasses many diverse areas of interest, to become highly philosophical in spirit. It is simply impossible to talk about the difficulties that students encounter when attempting to develop a skill that requires thinking in abstract terms without dealing with aspects of knowledge from many domains. Attempting to understand how people acquire a complex skill like programming implies taking a position on questions like how people see the world, how their particular viewpoint impacts on their ability to solve problems successfully, what the relationship between reality and knowledge is, and so on. So, in our case, as well as the core disciplines of computer science, education, and psychology, issues in science generally, epistemology, and even metaphysics come into the equation.

Some of this might seem to be irrelevant, even perverse, in terms of 'hard' science, but the truth is that everyone, even the most formal mathematician is, in reality, working within some philosophical framework even if they don't realize that they are so doing. The claim that some people make to a 'formalist philosophy', if such a thing can even be said to exist, leaves unanswered the question as to why it should be that their particular 'formal game' invariably appears to 'explain' aspects of the world in a way that most formal games, such as Chess, for example, don't.

> At heart, on a day-to-day basis, practically all mathematicians work in a highly intuitive fashion built on an out-and-out Platonist philosophy. Abstract mathematical entities such as numbers (natural, integer, rational, real, complex, infinite cardinal and ordinal) and spaces (geometric, metric, topological, linear, normed, etc.) are regarded as 'real objects' in a world that the mathematician sets out to *discover*. They are a part of a mental world that the mathematician learns to live in and become highly familiar with. Indeed, it is this intimate familiarity with an idealized, highly-ordered, abstract world of great simplicity that makes mathematics such an incredibly powerful tool with which to study certain aspects of the world.
>
> (Devlin 1995, p. 65)

While it is true that a good deal of computer science is founded in formal systems (hardware design, compiler grammars, etc.), and this is appropriate in strictly deterministic domains, it is also true that it exists, and deals with problems, in the real world. Any formal syntax, no matter how comprehensive or sophisticated it is within its own compass, must ultimately fail to capture the semantics of the real world, because "the Natural World is chaotic, its patterns stochastic rather than algebraic" (West 1997).

With Alfred Whitehead we see the role of philosophy as mainly contextual - to set each of the various pieces of knowledge that have been derived by reductionist analysis into the context of the others. Given that it is the case that each scientific enterprise necessarily selects and abstracts a particular set of relations from the world's complexity, it is the main job of philosophy to reintegrate them into a view of the world as a whole. "The task of philosophy is to recover the totality obscured by the selection" (Whitehead 1929, p. 20). This effort is inspired by the historic role of philosophy in questioning the accepted canons of the time, canons being a view of the world frozen in time. A steady unwavering gaze is important in terms of the particular but the whole consists of an infinity of particulars so understanding requires a Socratic style dialogue between the details (the science) and their interpretation (the whole).

Both rhetoric and the transmission of scientific knowledge are mono-

> logical in form; both need the counterbalance of hermeneutical appropriation, which works in the form of dialogue. And precisely and especially practical and political reason can only be realized and transmitted dialogically. I think, then, that the chief task of philosophy is to justify this way of reason and to defend practical and political reason against the domination of technology based upon science ... it vindicates the noblest task of the citizen ... decision-making, according to ones own responsibility
>
> (Gadamer, quoted in (Blacker) 1994)

In a sense, it is the fate of the concept of 'universe' to become watered down by the flow of immediate experience. The implication of the idea is that there is one unified basis for experience, that every single thing and event is fundamentally related, is a part of the 'universe.' But it is in the nature of subjective experience that one has to deal with those parts of the whole that are immediately apparent to the senses, William James' 'buzzing world' (Whitehead 1929, p. 68) of the present, overlooking the fact that the present is merely the past becoming the future.

> It lies in the nature of things that the many enter into complex unity. 'Creativity' is the principle of novelty. An actual occasion is a novel entity diverse from any entity in the 'many' which it unifies. Thus 'creativity' introduces novelty into the content of the many, which are the universe disjunctively. ... In their natures, entities are disjunctively 'many' in the process of passage into conjunctive unity.
>
> (Whitehead 1929, pp. 28-9)

The universe is a process, and process is, by definition, changing circumstance. Creativity is thus the mark of the universe, nothing remains the same for long - "it is inherent in the constitution of the immediate, present actuality that a future will supersede it" (Whitehead 1929, p. 305). So all the many different forms and interactions that we see are just passing glimpses of continuing and continuous creation.

> The fundamental inescapable fact is the creativity in virtue of which there can be no 'many things' which are not subordinated in a concrete unity. Thus a set of all actual occasions is by the nature of things a standpoint for another concrescence which illicits a concrete unity from those many actual occasions. Thus we can never survey the actual world except from the standpoint of an immediate concrescence which is falsifying the presupposed completion. The creativity in virtue of which any relatively complete actual world is, by the nature of things, the datum for a new concrescence, is termed 'transition.' Thus, by reason of transition, 'the actual world' is always a relative term, and refers to that basis of presupposed actual occasions which is a datum for the novel concrescence. An actual occasion is

> analysable. The analysis discloses operations transforming entities which are individually alien, into components of a complex which is concretely one. The term 'feeling' will be used as the generic description of such operations. We thus say that an actual occasion is a concrescence effected by a process of feelings.
>
> (Whitehead 1929, pp. 299-300)

This essential conjunction of 'the universe' and creative process is the source of the power of the pattern language idea. Here are all the elements of Alexander's formulation - patterns, process, wholeness and even the concern for aesthetics. "*Experience* involves a *becoming*, that *becoming* means that *something becomes*, and that *what becomes* involves *repetition* transformed into *novel immediacy* [emphasis in original]" (Whitehead 1929, p. 191). Patterns ('repetitions') are what we experience and the fact that they are all related in the unity of the universe (wholeness) means that they represent the 'datum' on which process ('becoming') occurs, producing the context ('novel immediacy') for the next 'experience' of 'becoming'. Understanding experience is therefore aesthetic appreciation (feeling) based on the 'logic' (details) of the interaction of the patterns of experience, pattern language.

> The human mind is the joiner, fitting together the disparate elements of the world to make objects, systems, sceneries. It can bridge distances from the size of an atom's nucleus to the space between galaxies, and leap over time spans of millennia as nimbly as over seconds. Contemplating the myriad isolated existences in the world of objects, my mind fits them all together into a universe. I remember or reconstruct what no longer exists and call it the past. I project or guess at what has not yet happened and call it the future. I connect the past with the present and invent purpose, a kind of nonlocal causality. I do the same with present and future, and create intentionality, also hope and fear. All of these are constructs of the mind, because neither past nor future exists in the world of objects. It is I who makes the waterfall. I gather up a hundred billion suns and make a galaxy. I link galaxies across vast spaces and plot their past and plot their past and future to the beginning and the end of time, and then wonder at the meaning of these limits. I perceive the juxtaposition of myriad atoms in a pebble and create its roundness, its colour, texture, its gestalt. In the language of quantum mechanics, I observe and measure, and cause the system I observe to fall into a definite state.
>
> (Harth 1993, pp. 9-10)

This is pure reason in the original sense of apprehending the reasons for experience being as it is. In this regard the philosophical impulse for coherence, surely a representation of wholeness, means that ultimately nothing rests on canonical authority - intrinsic reasonableness is the only test (Whitehead 1929, p. 53).

2.2 Pattern Theory in Learning to Program

We feel, then, that there is a conjunction between the activity of programming and Alexander's notion of "Pattern Language" in the idea of process. Doing anything involves a process and because it requires understanding, and understanding is itself a process, doing something is, at its most fundamental level, *understanding* something. So this is, also, the conjunction of philosophic and scientific endeavour, because scientific endeavour, at its most basic, is the revealing of detail, the collection of data, and, as Alexander points out, "the data of scientific method never go further than to display regularities. We put structure into them only by inference and interpretation" (Alexander 1964, p. 109). Strictly speaking, then, a lot of what goes on in scientific endeavour is actually philosophy, and this is probably why Einstein often referred to himself as a philosopher-scientist, insisting that "all religions, arts and sciences are branches of the same tree" (Einstein 1974, p. 9). He was, himself, never involved in the actual collection of the data, but in the *process* of understanding it. The fact of the constancy of the speed of light was established by the experimental work of Gauss, Ampère and Faraday, and was formalised in Maxwell's electromagnetism equations, what Einstein did was to consider the *implications* that flowed from that fact.

In programming, and in novice programming in particular, process is the critical factor. A major impetus of the present work is, thus, to examine the potential, on theoretical grounds, for basing a problem solving process, suitable for the first programming course, on a pattern language developed for that purpose. We feel that, in the migration of the pattern language concept into the computer science field, the emphasis has been on the reuse of proven solutions. Proven solutions, in programming terms, are previous experience packaged in pattern form. This seems, on the surface, to be the main power of the pattern idea for the novice programming situation. After all, what novices lack is exactly that, experience. However it has always been Alexander's contention that the concept of a pattern language offers much more than 'packaged' experience. His idea is that a pattern language is more than a simple collection of patterns in a particular domain. What transforms the collection into a language, is the relationships between the patterns. It is from this network of relationships that the process derives, and it is the process that reveals the fundamental properties that give order to designs, "order as becoming" as Alexander says (Alexander 2002*a*).

These elements have always been an important part of Alexander's thinking, but so far, as mentioned, the software pattern movement has concentrated on the problem-solution pairing of the pattern itself, and on the potential it provides for reusing solutions. So, in terms of patterns, and the use of patterns in the novice programming field, then, there is nothing particularly novel about this work. What is new here is the exploration and development of the *pattern language* concept in order to attempt to directly address some of the difficulties encountered in the learning to program situation. Other work has demonstrated that patterns are a useful way of presenting low level Java features to novice programmers by creating the patterns and using them in programming courses. In particular the work of Joe Bergin (Bergin 2005), Eugene Wallingford (Wallingford 1998), and others, with their 'Elementary Patterns' project has shown that the pattern idea can be extended down from the level of the advanced object relationship patterns that are the domain of the patterns introduced in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, Helm, Johnson and Vlissides (otherwise known as the Gang of Four, or GoF) (Gamma et al 1995), into the realm of basic programming language features.

It was on this basis that the current project began. In 2002, the first programming course, Computer Programming 1 (CP1) at this institution, Flinders Univerity of South Australia, was reorganised on the basis of using patterns. We had, some years previously, switched to Object-Oriented programming using Java, so the use of patterns in instruction was seen as a progression along the path of abstraction from the strictures of the machine environment. At the stage of introducing patterns into the first course, we had not yet developed the language notion fully in terms of pedagogical practice, so the patterns introduced in 2002 are merely a collection. In developing the patterns for the course, the topic co-ordinator, Dr Paul Calder, made extensive use of the resources of the 'Elementary Patterns' project mentioned above, and those in the textbook chosen for the topic, David Riley's "The Object of Java" (Riley 2002). He also chose to base the practical work on the BlueJ teaching environment (Kölling 2005) and to introduce pair programming (Williams & Kessler 2001) as an option in the practical sessions on the basis of research done in the Honours project that preceded this project. The assessment regime was based mainly on testing the student's knowledge of Java (70%), rather than programming ability as such, and this is discussed further in Chapter 10 where our attempts to measure the effectiveness of pattern languages in pedagogy are described.

The main thrust of this project has been in the development of pattern theory. Although the main element in this work is the incorporation of the pattern concept in the teaching of programming, it was clear from the start that the emphasis on process, in our case, the process of learning to program, has much wider implications in the use of patterns generally. The work at Flinders University in the pattern area has, through the exposure of an earlier project at the first two Australasian Conferences on Pattern Languages of Programs, come to the notice of Dr Jim Coplien, an early advocate of the spread of Alexander's ideas into the field of software development. It is through his involvement that we had access to early draft versions of Alexander's four volume work, "The Nature of Order" before publication. This has provided us with confirmation of the early emphasis on pattern process, because that is one of the directions in which Alexander is himself moving in the latest work, an exploration of what he calls, "the fundamental process".

It was the fact that this project was, for all intents and purposes, a continuation of an Honours project, that has meant that we have been able to build on a body of existing work, the core emphasis of which was the notion of a problem solving process based on the use of patterns. However, as the patterns developed for the earlier work were not organised into a fully fledged pattern language form, we found that this was inhibiting the development of the problem solving process. As the pattern language structure took firmer form, several issues that have been problematic in the software pattern idea have been made much clearer.

The pattern form is built around a name, a description of a problem, the context in which it occurs and the forces involved in causing it to be a problem, and a proven solution that solves the problem. It was found, when writing patterns during the earlier project, that the pattern components, "Context" and "Forces", were particularly difficult to compose. An early discovery of the current work, then, was that the difficulties in this area arose mainly from the fact that the patterns were developed in isolation from any notion of organisation into pattern language form. Once the work on organising the patterns into the pattern language structure began in earnest, many of the difficulties became less problematic. It turns out that the pattern language, the relationships between the patterns, is, in fact, the context of the patterns, and this is clearly demonstrated in the diagrammatic form of the language. The significance of the language for the content of the patterns themselves is the subject of Chapter 6.

Because the emphasis in the software pattern movement has been on the writing and using of patterns in the practical work situation, the power of the language idea has rarely been fully realised. Most pattern practitioners claim nothing more than that their work is a collection, or catalogue of patterns (see, for example, (Gamma et al 1995, p. 357)). This has resulted in descriptions of context and forces that are more general and ambiguous than is desirable, as we indeed found was true in the patterns developed for our earlier project. The drive to develop a process for the use of patterns, rather than just using them as individual design solutions, forced us to take the pattern language notion seriously, and doing this cleared up the problems we were having in actually writing the patterns themselves. Setting the patterns in the pattern language structure also makes it easier to organise the whole system in terms of the specific purpose for which it is being designed. In our case this has meant that imperatives of simplicity, clarity, and unambiguous process, that arise from the pedagogical purpose of the project can be reflected in the final form of the pattern language structure.

For example, while working on the patterns in isolation from the language structure we overlooked the flexibility in respect of the granularity of the system that the structure gives us. This caused us to expend effort on developing patterns at a far lower level than is desirable in the teaching situation. One of the big advantages of the pattern approach is the abstractive power that it provides. You are designing a solution in terms of the concepts that the pattern represents, rather than having to deal with the details directly. The structure imposed by the language form makes it clear that the point at which your system stops dealing with detail at the language level and begins to present it (hide it) within the pattern structure itself is a matter open to decision by the pattern language writer. Thus a lot of the lower level features of the Java language that were represented as patterns in our first pass over the system turned out to be material that is best presented within the pattern form. This means that the whole pattern system can be much smaller and simpler, and hence better suited to the pedagogical task for which it is designed. The detail is still all there, it is just more appropriately situated in terms of using the patterns in the novice problem solving situation. It turns up when the programmer needs it - in the final stage of the design procedure, the translation of the design into program code.

In essence, the pattern idea is just a way of looking at a system and because a system is a collection of related concepts organised in a way that is useful in terms of some purpose, it is driven by structure (order) and purpose (process). Therefore the design of the pattern view of a system needs to begin with a pretty clear view of the system in terms of its order and the process that gives rise to that order and enables the expression of its purpose. The patterns should flow from this structural analysis, and not, as is usually the case, the reverse. In our case the inherent order of the system we wish to build is the best possible organisation of Java resources for the solving of problems by novice programmers. Considering the project from this point of view enables the patterns to be designed to the purpose, rather than the purpose being retro-fitted to the patterns. This would seem to be a major shortcoming of current pattern practice in the software development field. In looking at Alexander's work it is clear that the purpose of his system, the design of living space with real human quality, is always at the forefront. The patterns are built around that purpose, never in isolation. But this does not seem to be generally true of the pattern thrust in the software arena. As Alexander, himself, said, while addressing a gathering of several thousand software development people at a conference in 1996:

> I think that insofar as patterns have become useful tools in the design of software, it helps the task of programming in that way. It is a nice neat format and that is fine. However, that is not all that pattern languages are supposed to do. The pattern language we began creating in the 1970s had other essential features. First, it has a moral component. Second, it has the aim of creating coherence, morphological coherence in the things that are made with it. And third, it is generative: it allows people to create coherence, morally sound objects, and encourages and enables this process because of its emphasis on the created whole.
>
> (Alexander 1999)

What the early stages of this project did was to demonstrate to us the force of Alexander's words. It was not until we began to see the patterns in terms of the whole system and its purpose, in our case, the teaching of programming, that the pattern view that we were trying to build began to come together. Alexander's message is that the order displayed in a system doesn't just happen, it is a result of the purpose of the system. The pattern concept reflects both the order and the process, and therefore is a mechanism for building, understanding, maintaining and using the order that is the system. It is a unique way of viewing a system because it doesn't just break the system down into its constituent parts, it provides a view based on the system in total, the inherent order that gives it coherence, makes it a system, rather than just a collection of related ideas, in fact.

In terms of programming, this manifests as a set of patterns based, not only on the features of the programming system (language), but on what might be termed the patterns of interaction with a real world system that is undergoing automation. Programming involves solving a problem that exists, not in the programming system itself, but in some other real world activity such as accounting, writing, building a compiler and so forth, so many of the design decisions to be made will turn on questions of a general nature such as repetition, swapping, choosing between several options and the like. So while the core patterns of a language for learning how to program will be, naturally enough, the features of the programming language being used, most of the connections between them involve general real world concepts, and some of these will manifest as patterns in the language.

Furthermore, code artefacts themselves often represent real world concepts, the program generated in Chapter 7, for example, produces a multiplication table, a common representation often found on the back cover of school exercise books, and, indeed, the GoF patterns can be regarded as real world concepts in this sense. This means that a program built using the constructs of a programming language, can itself, be a pattern in programming terms. As Salingaros says, "a coherent combination of patterns will form a new, higher-level pattern that possesses additional properties" (Salingaros 1998, p. 9). In this sense the code produced by the process delineated in Chapter 7 could be used in the Code Example section of a pattern for handling two-dimensional data as input or output, for example.

This "patterns made up of combinations of patterns" idea is a representation of the real world concept of "growth", and is itself, a pointer to the power of pattern languages in learning. If learning is anything it is the process of understanding - the *growth* of knowledge based on lower level concepts that you already 'know' - and the thrust of this dissertation is that learning is *the* classic pattern process; the design of one's mind, one could say, albeit design in a virtually unconscious sense. But, in fact, as we attempt to show in Chapter 9, most of learning, as with all other expressions of human 'expertise', is largely unconscious even though, at the conceptual level, knowledge is entirely self-referential in structural terms, one concept is almost always dependent on a connected web of other concepts, a concept map or pattern language, in other words. The human is, by nature, an "expert" at *learning*, it is what we spend our lives doing - generalising from specific experience, *patterning* life.

2.3 Patterns in Learning

Given the significance of learning in the human condition, it comes as no surprise that there are resonances in almost every field of intellectual endeavour, the fact of the genesis of the pattern language idea in Architecture being a case in point. But, although Christopher Alexander is the originator of Design Pattern Theory in the terms in which we understand it, it is important to notice that the idea of patterns in human existence is much older, and reference to this pre-design theory concern with patterns are made throughout this dissertation. The pre-design-theory literature, in essence, sees "patterns [as] those arrangements of internal relationship which give to any culture its coherence or plan, and keep it from being a mere accumulation of random bits. They are therefore of primary importance" (Kroeber 1923, p. 119).

Here we can detect all the features that Alexander ascribes to patterns, their primary role in how people live (culture), in creating wholeness (coherence or plan), in giving meaning to what would otherwise be "a mere accumulation of random bits", and the "arrangements of internal relationship" that make patterns out of collections of smaller patterns, a process to which we have previously alluded and which is the basis of the 'pattern language' idea. Kroeber even goes on to examine the correlation at the organic level. "What the present type of cultural pattern system shares with the fundamental organic patterns is that they both embody a definable system, in the repeated expressions of which, no matter how varied, there nevertheless is traceable a part-for-part correspondence, which allows each form or expression to be recognized as related to the others and derived from the plan as it gradually took shape" (Kroeber 1923, p. 123).

Alexander was trained in Mathematics and Architecture, and it is in this latter field that his work on patterns and pattern languages began. It seems to have been a response to the increasingly impersonal and dehumanised nature of buildings and the process of building design and construction. The notion of 'the quality without a name' would appear to be directed at making the human condition a consideration to be taken seriously in these fields.

> There is a central quality which is the root criterion of life and spirit in (all things). ... The search which we make for this quality in our own lives, is the central search of any person, ... it is the search for those moments and situations when we are most alive.
>
> (Alexander quoted in (Rising) 1997)

Something about the human sense of scale and feeling seemed to Alexander to be missing from the modern architectural practices and the buildings they produce. He argues that by failing to generate products that have human needs as their driving force, contemporary methods of design ultimately fail to improve the human condition and therefore to advance the art, and even the science, of design. The Alexandrian notion of patterns and, in particular pattern languages has, therefore, a sense of the place of the human in nature.

> Yet, changing as it is, each language is a living picture of a culture, and a way of life, The patterns it contains, widely shared, reflect a common understanding about the ways that people want to live, the way they want to rear their children, the way they want to eat their meals, the way they want to live in families, the way they want to move from place to place, the way they make their buildings look towards the light, their feelings about water, above all, their attitudes to themselves. It is a tapestry of life, which shows, in the relationships among the patterns, how the various parts of life can fit together, and how they can make sense, concretely in space. And, above all, it is not just a passive picture. It has a power in it. It is a language, active, powerful, which has the power to let people transform themselves, and their surroundings.
>
> (Alexander 1979, pp. 347-8)

The big pattern, then, is this relationship between the patterns in nature and the way that human beings think, and therefore, act - what might be called the human pattern. Alexander continues to work on developing his theories in terms of their application in understanding and improving the human condition. The interesting thing here is the centrality of the pattern language idea in Alexander's design philosophy. In itself, too, the idea of a language of patterns is quite novel, even unusual. Normally we think about a language in terms of expressing meaning by means of a vocabulary and a set of rules, the language grammar, for putting the words that make up the vocabulary together. Alexander's idea is that the patterns which represent the problem-solution pairings in a domain form a structure by way of the relationships between them, and that this structural coherence implies a sort of meaning. This web of 'meaning' allows the concepts embodied in the patterns to be put together in sequences to express larger and more complex concepts just as the words in a language do. In fact, what Alexander is proposing in the quote above is patterns as a "language of thinking" by which we create "understanding", an idea that we examine in detail in Chapter 5.

The volume, "A Pattern Language" is the presentation of Alexander's thinking about the built environment in the form of a pattern language containing 253 patterns. This form enables him to explicate the pattern idea from within and beyond the patterns themselves. This is important because the pattern has two aspects, it is powerful in two ways. Firstly, internally, it encapsulates previous experience through discussion of the problem, its context and the forces applying in the situation that are causing it, and the solution. Secondly and most importantly, the book examines the relationships between the patterns that make the collection a coherent whole, a form which Alexander calls a language. Like a word, each pattern only expresses a single simple concept. Being part of a larger entity, the language, enables the smaller concepts to express any concept, no matter how large, by combining concepts together in sequences.

In Alexander's case this involves demonstrating not only how the patterns work in a single building project, but how the same ideas can be used on a larger scale in town or district planning, to give coherence to an otherwise disjointed system. Alexander's whole point is that the current methods fail in a holistic sense because they ignore the human dimension in designing living spaces. Unfortunately the holistic aspect, which Alexander expresses in terms such as wholeness and 'the quality without a name', has become, somewhat controversial, and has been used to deride his thinking as un-scientific. But all he is saying is that any process which concentrates only on the physical aspects of the built environment, and ignores the fact that this is a place where humans are to live, misses the point that a built environment has biological, psychological, social, and spiritual implications as well as the physical ones. No building, or town, exists in and of itself. Its purpose is, or *should be*, to fulfill human needs, and its design, therefore, needs to be cognizant of all facets of the human condition. It is the counter argument, that buildings etc. can be designed in terms of physical space only, ignoring human space, that is un-scientific, because these things are human living spaces not mere physical monuments to some designer or owner.

In terms of design pattern theory "A Pattern Language" (Alexander 1977) is the foundation. The basic elements are all here, the patterns themselves, the components that make up the internal form of the pattern, the relationship between the patterns that form the network that Alexander terms language, the generative process that enables sequences of patterns to be formed, and the human framework that supplies the moral and spiritual direction that enables wholeness. Because the theory is being explained in terms of the patterns presented in the book, there is a practical down-to-earth feel to the presentation of theory. The patterns are the focus, and the theory is built around them. This means that some things, like the details of process and the working out of the language idea, are left for later consideration. As Alexander himself has said there seemed to be something more fundamental that was missing from the pattern language (Alexander 1999, p. 81). These considerations however are more about why the theory works, and how it works. In terms of applying the theory, "A Pattern Language" is the fundamental guide to practice. Of course, it is written in terms of designing the human living environment, and its application is directed at architectural practice. Therefore migrating design pattern theory to other domains will necessarily involve adapting the basic elements, the patterns, the pattern language, the generative process, and the human framework to the new context. Because the theory works, the software community has tended, perhaps like Alexander in his field, to take the theory aspect for granted. Nevertheless, in practical terms, "A Pattern Language" is the basic primer for all design pattern practitioners, no matter what field they are in.

"The Timeless Way of Building" (Alexander 1979) is the theoretical background for patterns. It takes the theory from the first book and elaborates and extends it. In terms of the theory there is very little that is new, but rather its purpose is to explain the material that was in the first book. It attempts to remove the theory from the overwhelming presence of the 253 patterns, and while this makes for a more condensed and concentrated exposition of the theory, it loses most of the practical immediacy that "A Pattern Language" has. This probably means that, despite the fact that this work is labelled as Volume 1, it is important to read these two works in the order that they were produced, as, on its own, the second book is likely to be somewhat obscure without the full force of the pattern background. Having said that, this work is important for the migration of the theory into other domains, precisely for those reasons. The theory, here, is partly isolated from the particular context in which it arose, and it is therefore easier to begin the transition.

The central thesis of this book is that, until modern times, the design of the human environment has always been in the hands of those who lived in it. There was a direct relationship between people and the structures they built and lived in.

> The order of a building or a town grows out directly from the inner nature of the people, and the animals, and plants, and matter which are in it. It is a process which allows the life inside a person, or a family, or a town, to flourish, openly, in freedom, so vividly that it gives birth of its own accord to the natural order which is needed to sustain this life.
>
> (Alexander 1979, p. 7)

By directly reflecting the life of its designers the built environment achieves the timeless quality which Alexander refers to as 'wholeness'. It was in studying the structures built in this way that he discovered the patterns and formulated his pattern language theory.

> The people can shape buildings for themselves, and have done it for centuries, by using languages which I call pattern languages. A pattern language gives each person who uses it, the power to create an infinite variety of new and unique buildings, just as is ordinary language gives him the power to create an infinite variety of sentences.
>
> (Alexander 1979, p. xi)

The implication of this idea is that the patterns are a repository of design solutions proven over time. They are not just a reflection of a passing series of fashions, or of changes in building technology, because they have endured. The architectural pattern language of a particular society represents the accumulated 'wisdom', in the full sense of the word, of a way of life in relation to its environment, just as the genome represents the accumulated 'wisdom' of a way of life for biological organisms in relation to their environment. And, again like the DNA, being a language, it is also a way of expressing that knowledge into the future.

"The Nature of Order" (Alexander 2002*b*) is a vast four volume work to which we have had access to early draft versions of the first two volumes thanks to the agency of Dr Jim Coplien. This work is an attempt to build the framework for an understanding of the properties and processes that underlie order in nature. It grew out of the earlier work on patterns because Alexander saw that, in practice, the techniques that he had advocated were not fulfilling his expectation of them.

> I began to notice a deeper level of structure and a small number (15) of geometric properties that appeared to exist recursively in space whenever buildings had life. The 15 properties seem to define a more fundamental kind of stuff; similar to the patterns we had defined earlier, but more condensed, more essential - some kind of stuff that all good patterns were made of.
>
> (Alexander 1999, pp. 75-6)

Therefore this work is more of an explanation of why patterns work than how they are applied. Nevertheless it is important to pattern practice because it expands and clarifies aspects of the earlier work. For example, whereas the generative process was a constant theme of pattern theory in the earlier works, here it is examined much more deeply, with the purpose of helping to set pattern theory into the larger context. This context is how the 15 properties give rise to geometric centres in space. It is these centres that underlie the symmetries of patterns that we see. But more importantly it enables us to understand how symmetry breaking is a structure preserving process which allows the unfolding of design that enhances the order we see in nature rather than working against it. Patterns are seen here as a reflection of the underlying order. Like Plato's shadows on the cave wall, the patterns are a view of reality, rather than reality itself. And like Plato's shadows, they enable us to begin to understand reality. The scope of "The Nature of Order" is, thus, an attempt to clarify our understanding of order, to show that, in essence, it is ultimately the order that we discern in nature that gets expressed as the quality of 'wholeness' in human structures.

> For I believe it is the nature of order itself, which has soaked through with I. The essence of my argument .... is that the I, the thing I call I, which lies at the core of our experience is a real thing, existing in all matter, beyond ourselves, and that we must understand it this way in order to make sense of living structure, of buildings, of art, and of our place in the world.
>
> (Alexander quoted in (Salingaros) 2002*a*)

Patterns have been a continuing theme of Alexander's life. His PhD thesis, published as "Notes on the Synthesis of Form" in 1964 (Alexander 1964), began the journey that led to pattern theory. His argument here, was that our means of handling problems does not scale up well with the increasing complexity of the situations in which problems must be solved. For example as residential areas grow they increase the demand for heavy transportation which tends to overrun the ability of the transport system. The simple solution to increase the size of the transport arteries conflicts with the demand for calm and safety in residential areas and interferes with other established service routes, such as those for water, gas, electricity and lines of communication. But more complex solutions raise issues of cost and environmental concerns. But worse still is the fact that the complexity causes the problem to cut across domain boundaries, meaning that the expertise needed to consider solution designs cannot be found in one place, and the need to consult across areas of expertise only adds further complications to the design process. The answer that Alexander posits is a design revolution, analogous to the earlier industrial one. Just as machinery was used to magnify the human physical capacity, so now we need a means of magnifying our design inventiveness. Science has over the years developed a way of thinking of shape (form) in abstract terms called mathematics.

> The shapes of mathematics are abstract, of course, and the shapes of architecture concrete and human. But that difference is inessential. The crucial quality of shape, no matter of what kind, lies in its organization, and when we think of it this way we call it form. Man's feeling for mathematical form was able to develop only from his feelings for the processes of proof. I believe that our feeling for architectural form can never reach a comparable order of development, until we too have first leaned a comparable feeling for the process of design.
>
> (Alexander 1964)

The ideas that culminate in "The Nature of Order" are all present in this early work. The history of the subsequent years shows that they have not, by and large, been adopted in design circles, and, in particular, in Alexander's own field of Architecture. In some ways, his ideas have fallen on more fertile ground in Computer Science, probably because here, the complexity of systems has even further outstripped the abilities of the human mind to comprehend them in totality. Some of the best pieces about what might be called the philosophy of patterns, have been written by James Coplien. His "Space: The Final Frontier" (Coplien 1996*b*) starts with Alexander's theories, particularly the theory of centres, and tackles the issue of aesthetics, an aspect of Alexander's thinking that is avoided by most authors in the software domain. In "The Geometry of C++ Objects" (Coplien 1998), he explores the relationship between Alexander's theory of centres and the pattern language idea to begin to grapple with the idea of geometries in software. This is an important issue because the idea of centres and symmetries are major themes in Alexander's current thinking. Translating them across from the physical world to software is turning out to be a difficult task, and one that will assume greater significance now that "The Nature of Order" is more widely available.

It seems to us that one of the areas that was neglected during the transfer of the pattern concept is the difference between the design spaces involved. Forces and geometry, for example, are much clearer ideas in the physical space of building design than they are in the metaphorical 'space' of software. In "Close the Window and Put it On the Desktop", Coplien examines the place of metaphor in understanding, learning and design, and concludes that "learning is not a separable, detached element of design. It is intrinsic to design. It may be fundamental to design" (Coplien 2000*a*). Although this is written in relation to using and learning to use software, his point about "experientially based learning" applies in learning to develop software as well. The report of a 1997 address by Coplien "On the Nature of The Nature of Order", written by Brad Appleton (Appleton 2000*a*), brings us back to Alexander, and stresses the importance for software pattern practitioners to go back to the roots of the concept.

Of the works specifically about Alexander, Linda Rising's "The Road, Christopher Alexander, and Good Software Design" (Rising 1997), and "Some Notes on Christopher Alexander" by Nikos Salingaros (Salingaros 2002*b*), are good introductions. Doug Lea's "Christopher Alexander: An Introduction for Object-Oriented Designers" (Lea 1993) is another good piece in this mould. But probably the best exploration of Alexander's ideas is the biography by Stephen Grabow, built largely around hundreds of hours of discussion with the architect, and which contains long quotations taken from these discussions. In the technical aspects of patterns in software design, all of Jim Coplien's many works, particularly, the comprehensive treatment in his 1996 book, now out of print but available on the Web, "Software Patterns" (Coplien 1996*a*), are well worth reading. He manages to cover a breadth of detail in the field matched by few others, including his forays into the area of symmetry and symmetry breaking (Coplien 2001) (Coplien & Zhao 2000). Another article by Salingaros, "The Structure of Pattern Languages" (Salingaros 1998), provides a good overview of the connective power of patterns - Alexander's pattern language concept.

In terms of particular patterns, the patterns in the original Gamma book are widely covered elsewhere in the literature. James Cooper's book on the Java version of these patterns, "Java Design Patterns: a Tutorial" (Cooper 2000), is particularly useful for programmers in this language. The best quick introduction on the web to these patterns is in the Object Design Workshop by Bill Venners, "Designing with Patterns" (Venners 2002). In terms of a general introduction to the pattern concept, Brad Appleton's web pages "Patterns and Software: Essential Concepts and Terminology" (Appleton 2000*b*) and "Patterns in a Nutshell: The 'bare essentials' of Software Patterns" (Appleton 1999), are excellent places to start. However because of the ubiquity of the pattern concept these days, many books about other aspects of software engineering and programming contain short introductions to patterns. Jeff Nelson's "Programming Mobile Objects with Java" (Nelson 1999) is a good example of a work about something else entirely that has a good general overview of the pattern idea, and, of course, most contemporary software engineering texts would be similarly endowed, the classic Pressman (Pressman 1997) being a case in point. The Patterns Home Page (Hillside 2005) contains a multitude of links to pattern material, as does Brad Appleton's Software Patterns links (Appleton 1998), and Wiki Wiki Web's repository (WikiWikiWeb 2005).

2.4 Patterns in Education

As Locke pointed out almost all knowledge is gained by an interaction between experience and reason (Locke 1910). Education is therefore a process directed at imparting knowledge, but as it becomes institutionalised the tendency is more and more towards rote learning, it tends to become one experience rather than a wide variety of experiences. John Dewey, who was one of the foremost influences on education during the last century, based his philosophy on providing the student with as wide a variety of experiences as possible. From democratic principles Dewey drew the notion of the student having control over the learning experience. It follows from this that knowledge is active, it arises from the discovery and testing of ideas rather than their static presentation. If, in dealing with a problem, the student "cannot find his own way out he will not learn, not even if he can recite some correct answer with one hundred percent accuracy" (Dewey 1966, p. 160). Two conclusions important for education follow.

> (1) Experience is primarily an active-passive affair; it is not primarily cognitive. But (2) the measure of the value of an experience lies in the perception of relationships or continuities to which it leads up. It includes cognition in the degree in which it is cumulative or amounts to something, or has meaning. In schools, those under instruction are too customarily looked upon as acquiring knowledge as theoretical spectators, minds which appropriate knowledge by direct energy of intellect. The very word pupil has almost come to mean one who is engaged not in having fruitful experiences but in absorbing knowledge directly.
>
> (Dewey 1966, p. 140)

This is extremely pertinent to the learning of programming, of course, because programming is inherently a practical skill. But this epistemological focus on the importance of combining different perspectives, of having multiple learning experiences, is used, by Gregory Bateson, as the basis for examining parallels between learning and evolution as stochastic processes, and for constructing a general purpose epistemological schema consisting of a switching back and forth between form and process (Bateson 1979), reminiscent of the iteration between the problem solving process and the pattern language structure outlined in Chapter 7. The concentration on learning as an active task-based phenomena, also leads to the idea of minimalist instruction. This idea, that concentration on specific tasks within a field produces learning outcomes that are more effective in a shorter time, which emerged during the 1980s, is an attempt to bring theory, research, and practical design experience together in order to elucidate design practice (Carroll 1996).

All this focus on practical knowledge bears on the problems that arise when teaching programming. What we see constantly, as educators, is the lack of a sense of process (what am I supposed to do?) in students when faced with a problem. We don't really care about the actual 'knowledge' that a pattern encapsulates, what we are interested in are the handles it provides on the problem-solving process itself. That's also why we don't care about the technical issues concerning the definition of these things. Patterns or idioms or thing-a-me-bobs doesn't matter. We are using these things as patterns, therefore they are patterns, as far as we are concerned. The philosophy is; give students the tools (the patterns, the language and the process), point out the features that patterns add to the problem-solving situation, and let the learning start. We can't 'teach' them how to solve problems using patterns directly because there is no consistent method to the process, each solution-process is going to be different! But the pattern language concept provides an overarching meta-process based on the building up of a sequence of patterns, and it is this meta-process that is the source of its power in educational terms.

This meta-process, described in Chapter 7, drives the *actual* process by which a particular problem is solved. The *actual* process derives from applying the meta-process of identifying the patterns required and building the sequence that solves the problem. Our description probably looks rigid and mechanistic, most, if not all, descriptions of process look thus, which is precisely why maps, rather than descriptions, are more instructive. But the chapter is about demonstrating that the pattern idea should have some usefulness at this level, not as a description of how patterns will be used in general. If we were describing the process of solving a different problem it would look quite different! Patterns are, by nature, a minimalist approach to 'teaching'. Nobody sets out to explicitly 'teach' patterns, as such, what is required of patterns in the novice programmer's design space is the release of natural creativity by providing a place to start thinking about the problem to be solved.

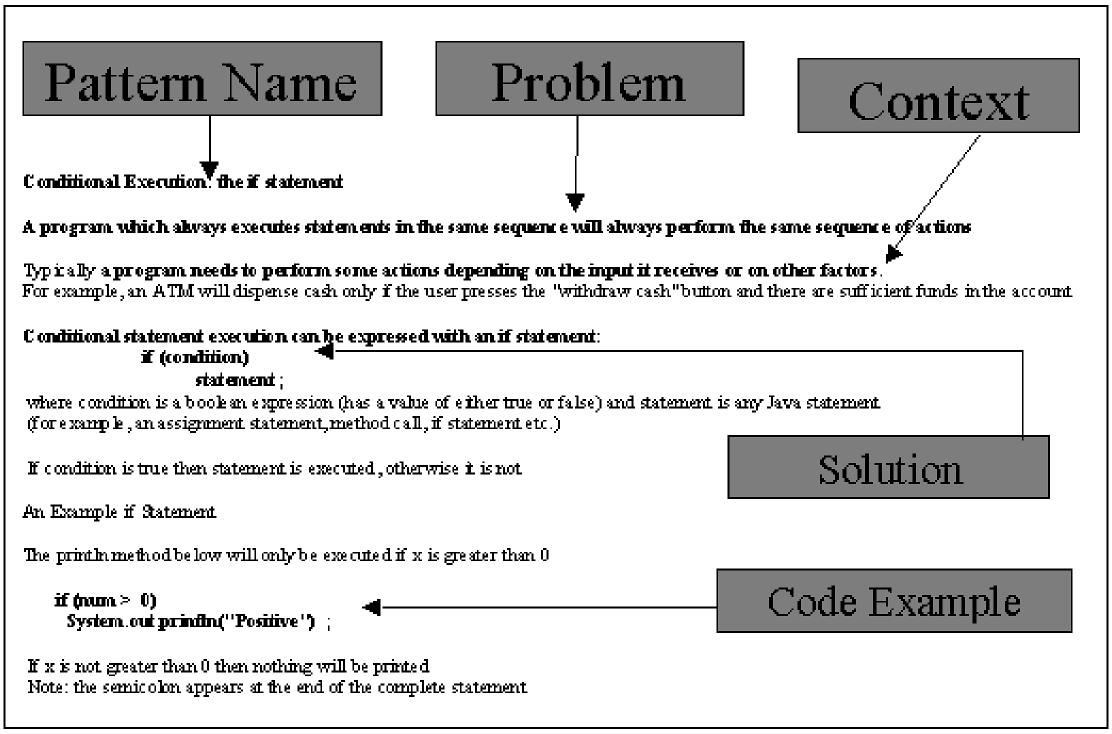
This fits with Bloom's notion of cognitive levels of learning, the idea of a taxonomy of what educators want students to know encompassed in statements of educational objectives (Bloom 1971). Patterns, using the term in the general sense, can be seen to be important at all levels in the cognitive process, but in the programming context, and in the area of the software pattern concept itself, clearly the analysis and synthesis processes are critical. The aim of the analysis of a problem situation is to discover the patterns revealed in the distribution of the forces and constraints within the situation. Of course this needs to be done on the basis of a knowledge and comprehension of the programming resources available in the domain. The creative part of programming involves matching the results of the analysis of the problem with the appropriate resources to build the new synthesis of the forces that solves the problem. Because this is a pattern matching process, identifying the patterns in the situation with the patterns of syntax available in the programming environment, the idea of presenting the resources in a specialised format to assist the matching process arises. And because of the association with pattern matching, this specialised format has come to be known as a pattern, in the specialised sense of design or software pattern.

Another way of looking at the results of applying Bloom's taxonomy to the programming domain is that it points to the practical aspect of the field. The design patterns show up in the categories that are most clearly involved in the programming activity itself. In learning this activity, patterns then will be mainly useful in that they can be seen as a way of encapsulating knowledge in a form useful to the activity. This is, probably, the most powerful aspect of design patterns in the purely pedagogical sense. It is doubtful that patterns have much pedagogical power in themselves in terms of driving a problem solving process, this is the role provided by the relationships between the patterns, the language. But for the practical aspect of solving programming problems, one would probably not bother to write programming language constructs into the design pattern form.

So, insofar as a pattern language is to be used in the teaching context, it is pedagogical in nature. However, I don't think we see its main use as a teaching tool or methodology. Our view is that you simply present the patterns in the teaching material in the places where examples are normally used. One of the 'discoveries' made in this project, is that all the elements of a pattern are there in the examples used in course material already. They are just not presented in a systematic way. An example from the past CP1 lecture notes at Flinders University that illustrates this is presented below. The various elements of the pattern form are highlighted in bold text, and identified.

Therefore, the idea of systemizing the examples in pattern format is to provide the extra benefits of the pattern form in the problem solving situation, not just to help students learn the material. The skill of solving problems grows out of practice, and patterns are directly useful at this level. Examples are important here, but novices mostly fail to pick up their 'essence'.

> This is a bone of some contention in the wider patterns community (that is, wider than a single domain). There are those who staunchly uphold that abstraction is at best a meaningless aim and at worst a dangerous one; that complex things are complex and that abstracting away from that complexity makes a false goal of simplification. (Gabriel 1996*b*) But, as pedagogues, we know that abstraction is a very difficult step to take (Bloom 1956); that learners find it difficult to grasp the principles embodied in a single example (or a series of single examples) then isolate it as the common referent they all share (that is, abstract from the details to the principle) and apply that principle in novel situations. In equal fashion, novices appear to have


Pattern Name
Problem
Context
Solution
Code Example

Conditional Execution: the if statement

A program which always executes statements in the same sequence will always perform the same sequence of actions

Typically a program needs to perform some actions depending on the input it receives or on other factors.
For example, an ATM will dispense cash only if the user presses the "withdraw cash" button and there are sufficient funds in the account

Conditional statement execution can be expressed with an if statement:

```
              if (condition)
                     statement ;
```

where condition is a boolean expression (has a value of either true or false) and statement is any Java statement
(for example, an assignment statement, method call, if statement etc.)

If condition is true then statement is executed, otherwise it is not

An Example if Statement

The println method below will only be executed if x is greater than 0

```
    if (num >  0)
      System.out.println("Positive")  ;
```

If x is not greater than 0 then nothing will be printed
Note: the semicolon appears at the end of the complete statement

Figure 2.1. Extract from CP1 lecture notes, Semester 2, 2000 (Roberts 2000)

a poor grasp of detail and little appreciation of what is (and is not) important in a given situation. Both of these extremes are addressed in a good pattern, which must abstract to a quality from a set of examples and at a level which is immediately graspable. If it is too abstract, then it will not be apparent; if it is too concrete, then it will not be perceived as separate from the detail of the examples themselves. Finding the correct level of abstraction is a teaching skill; it is also a goal of the best patterns.

(Fincher & Utting 2002)

One of the main difficulties of examples for novices is they attempt to apply them in a rigid manner, failing to grasp the general principle that lies at the heart of the example. Patterns would appear to express the general nature of the problem-solution nexus much more clearly, and, moreover, they provide 'slipperiness', that is they make the example more easily exportable from one context to another, so that it can move across domains like analogies do in natural language. Design patterns are, to all intents and purposes, analogies in the programming domain. In a sense they are attempting to highlight the general nature of the example - to hide the code in effect. Seeing 'the code' as 'the example' is a large part of the difficulty in applying it in similar, but different, situations. The pattern form sets 'the code' in a more realistic context - generalizes it, in effect.

There are many discussions about the difficulties involved in learning to program, and the possible causes and cures. This literature predates patterns and has been one of the circumstances behind the continual changes in the Computer Science curriculum. Of course, the rapid pace of change in the computing environment has itself been the principle force. It has caused the continued and continuing need for the computer science curriculum to attempt to reflect the changing environment. This need for the curricula to keep up has long been recognised, and in order to provide a set of standards for considering the issues involved in meeting the needs of industry in computer science education, the IEEE and the ACM both have produced a series of computer science curriculum reports over the years. The ACM series began in 1968 and the original report was updated in 1978 and 1991, and the IEEE produced reports in 1983 and 1986 and joint reports with the ACM in 1991 and 2001. All of these reports consisted of a general discussion of the issues involved in the contemporary situation and recommend various changes to the computer science curriculum. And the difficulty with the teaching of programming is a recurring theme.

The 1989 publication, "Studying the Novice Programmer" (Soloway & Spohrer 1989), is a collection of articles on the learning to program issue. Some of these are along the lines of using programming to teach problem solving in mathematics and the like, an interesting reversal of the idea behind this project. However many of the rest focus on the psychological aspects of programming, and discuss things like concept acquisition and use, which resonate with the pattern idea. The discussion, "The Tasks of Programming", moderated by J. Herbert, concentrates on the problem solving aspect, and states that "expert programmers understand the problem by drawing on past experience, but novice programmers have very little", and must therefore start from scratch each time (Herbert 1997).

Understanding code written by others is an activity that is recommended for novices, and this is studied in Brooks, "Towards a theory of the comprehension of computer programs" (Brooks 1983) and in Wiedenbeck, "The initial stage of comprehension" (Wiedenbeck 1991). Program comprehension is another one of those activities that relates closely to the pattern idea of reusing expertise. One of the motivations for the introduction of OO programming was the idea that it is a better environment in which to model real world entities and activities. It was expected that this would be an advantage in the novice situation but a comparative study of program comprehension in novices in the OO and procedural paradigms, concluded that the trade off between the increased complexity of the programming language and the intuitive fit of the OO paradigm with the real world, was not clear cut (Ramalingam & Weidenbeck 1997).

Given the success of software patterns in advanced programming in industry, it is not surprising that they have had considerable impact on Computer Science Education. The pattern idea was quickly adopted in Software Engineering type courses due to the obvious reuse implications, but the use of patterns in programming type courses has been slower to develop. The motivation for the

Software Engineering discipline has always been to develop a set of methods for developing software to a consistent standard of performance and reliability, and the reuse of code and experience has always been a factor in these considerations. However the early experience with software reuse was not completely successful. In 1992, a survey on software reuse found that it was “not living up to its original expectations”, and concluded that “abstraction plays a central role in software reuse. Concise and expressive abstractions are essential if software artefacts are to be effectively reused” (Krueger 1992).

The design pattern concept fitted into this need for abstraction in the reuse technique, and patterns emerged as a “software engineering problem-solving discipline” (Coplien n.d.). As Pfleeger says in one of the discipline’s standard texts, a design pattern “names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable ... design” (Pfleeger 1998). However the significant implication of the reuse idea as expressed in design patterns is the reuse of expertise. What a design pattern represents is a proven solution to a common problem. This is reusing ideas rather than elements, artefacts, or even principles, in the engineering sense of reuse. And the reuse of expert experience is an educational process in itself.

The initial enthusiasm of the Software Engineering community for patterns seems to have faded recently, with many authors pointing out perceived shortcomings in Software Engineering terms. Some of these, like worries about pattern language ‘completeness’ (Mattson 1996) and patterns carrying bad ideas as well as good ones (Crawford 1996), are due to the incomplete understanding of pattern theory caused by the too casual transition into the software field. Other criticisms like the finding that “integrating patterns into the software development process is a human-intensive activity”(Mattson 1996) are caused by unrealistic expectations. The most interesting critique is that patterns are only a learning aid - “when designers become experts they discard patterns”(Mattson 1996) - which is, of course, the feature that is of most interest to educators.

Many instructors have noticed the pedagogical implications of the pattern language idea, but a lot of the articles that deal directly with the use of patterns in programming courses seem to imply that introducing patterns should be left to the second programming course, normally a data structures based topic. Probably this is because these articles are based on the idea of using existing patterns at the level of those in the (GoF) book (Gamma et al. 1995), rather than on patterns for lower level programming language constructs. Gelfand (Gelfand et al. 1998) discusses the use of GoF level patterns such as adapter, comparator, decorator, iterator and locator, and demonstrates the use of these patterns to, for example, adapt a dequeue class so that it can be used to implement the stack abstract class using the adapter pattern, and to generalise the implementation of an algorithm that uses, say, a simple tree traversal, using the template pattern so that it can be used for several different purposes. This approach is based on the idea of introducing the design principles inherent in the patterns as early as possible in the computer science curriculum. However, there is no discussion, or even acknowledgement of, patterns for programming language constructs below the level of classes or objects, the two specification items used in the article, hence this article does not much enlighten the task being undertaken in this project.

In this regard the 1999 paper, "Design Patterns for the Data Structures and Algorithm Course" by Preiss (Preiss 1999) is similar. It too deals with the use of patterns from the GoF book and is therefore not directly relevant to the idea of using patterns at a lower level. However, although based on ideas used in the CS2 course at Waterloo University, it is more of a theoretical piece than the Gelfand article, which is predominantly a discussion of the patterns used in a particular course. Preiss discusses concepts like the relationship between the abstraction aspect of patterns and design.

> Design patterns are emerging as the abstractions that are appropriate for talking about designs. They provide a framework for thinking about and for comparing design decisions and trade-offs. More importantly they provide a common vocabulary for describing software designs.
>
> (Preiss 1999)

Preiss believes that patterns provide a way of linking apparently unrelated ideas and are thus powerful as design and problem solving tools in the teaching of programming.

The lecture notes for the introductory programming course in C++ at Brown University, CS1, explicitly introduce four patterns. These are the state, proxy, chain of responsibility, and factory patterns from the catalog of patterns in the GoF book. While these are useful patterns to be presented early in a novice programmer's career there is nothing in this material to suggest that patterns for lower level programming language constructs even exist. That is, this course follows the line that "patterns identify and specify abstractions that are *above* the level of single classes and instances, or of components [emphasis added]" (Gamma et al. 1993, quoted in (Cooper) 2000, p. 5), and doesn't attempt to develop or present patterns below this level. The four patterns presented in this course are clearly some of the easiest of the set in the GoF book, and are therefore the most appropriate of these patterns to be used in this context. However the fact is that the opportunity to develop and use patterns at a lower level than these is missed.

2.5 Patterns In Introductory Programming

There is a great deal of literature on the problems encountered in acquiring programming skills (Denning 1989), far too much to cover adequately here, so this section is meant more as an overview of some interesting ideas that relate to the objectives of this project, than as a comprehensive review. Many papers point to the lack of explicit material dealing with design issues at an early stage in programming courses as a contributing factor to difficulties with solving programming problems (McKeown et al. 1999). Object design, in particular, is mentioned by several authors as a skill that should be inculcated early in a novice programmer's career (Biddle & Mercer 1997) (Northrop 1992). Most of these articles do not relate these concerns specifically to the use of patterns in the programming curriculum, as many were written before the pattern concept entered the software development field. And, of course, the real issue here is not a lack of the requisite knowledge, per se, but rather the difficulty in applying it in novel situations, "the ability to isolate concepts from any of the examples that give rise to them" (Skemp 1971). So even in these works written before the pattern idea arose, it is possible to discern the pattern concept in form if not in name. Thus in Robert Floyd's 1979 article about his 'paradigms of programming' concept, it is clear that he is talking about something, "a general rule for attacking similar problems", that is very similar to patterns.

> After solving a challenging problem, I solve it again from scratch, retracing only the insights of the earlier solution. I repeat this until the solution is as clear and direct as I can hope for. Then I look for a general rule for attacking similar problems.
>
> (Floyd 1979)

Similarly in 1986, Elliot Soloway was talking about "breaking down a problem on the basis of sub-problems for which you have canned (or almost canned) solutions" (Soloway 1986). The breadth of this material indicates that the lack of early exposure to design principles is widely seen as a problem in introductory programming courses. From this depth of concern it would seem that the idea of investigating the usefulness of using patterns in the learning process warrants some attention in terms of providing the "relevant schema for solving problems" (Wallingford 1996, p. 28) that students have "traditionally ... been left to construct" (Wallingford 1996, p. 28) implicitly. And especially so given the close relationship that already exists between the design pattern movement and object oriented design that is cited as a particular problem in many papers.

The most comprehensive and most interesting of the work about patterns at the level dealt with in this project is the material that appears on, or is linked to, the web site of Joseph Bergin of Pace University and the "Elementary Patterns" Home page of Eugene Wallingford at the University of Northern Iowa (Wallingford 2001). It is difficult to characterise this material as some of it is not explicitly course material, but neither is it all in the form of academic papers in the normal sense. Many of the pages are presented as groups of patterns, organized around a single programming concept, such as "Patterns for Selection" or "Loop Patterns", or around a level of programming skill, for example, "Coding at the Lowest Level: Coding Patterns for Java Beginners" (Bergin 2002) and "Simple Design Patterns"(Bergin n.d.). However some of the pages represent papers about patterns, or about using patterns such as"Moving towards Object-

Oriented Programming and Patterns" and "Patterns of Object-Oriented Design for Novices". And there is even some material that Bergin calls"Pedagogical Patterns for Teaching Computer Science", pages such as "A pattern for Teaching Patterns" and "Teaching Objects with Elementary Patterns"(Bergin 2005).

Thus the mass of material on these sites contains much that relates directly to this project, and was consulted regularly during the setting up of the new CP1 course here. Some of Bergin's patterns equate directly to patterns used here and some of his thinking has been incorporated into these. Perhaps the main weakness of the site, as it relates to using patterns for introductory programming, is the lack of a sense of organization. The patterns are probably all here, but finding a particular pattern or patterns that relate(s) to a particular programming situation could be rather problematic, unless you were fairly familiar with the site already. Bergin himself does not claim this material to be anything other than a work in progress, so the above statement is not meant as criticism, but rather as an observation about incorporating aspects of it into the introductory programming course situation. That said, this site should be given a link in the resource section of any programming or software design course. There are many insights here that are too useful to students who are thinking deeply about programming and design issues to wait for them to be formally incorporated into programming courses. Given the ubiquity of pattern methodology in software development these days, the exposing of students to pattern thinking early in their programming careers has to be an advantage.

But the most powerful aspect of this material is the light it casts on the difficulties that novice programmers have in learning to program. It is obvious from reading the programs produced by many students that, despite the fact that the courses are organized around an OO language, Java, they are learning procedural programming still, as others have pointed out (Weber-Wulff 2000, p. 85). Bergin's material might help to explain why this is so. His contention is that although the paradigm being taught has changed, the methods of presenting it have not. The patterns appropriate to the old paradigm are still being used. "In top-down programming you conceive of the program as a whole as a process to be discovered. You think of the problem as a whole as a process to be decomposed" (Bergin 2000). Whereas in the OO context the approach required is quite different, involving a completely different mindset.

> The methodology for constructing an object-oriented program is to discover the objects first via a process of simulation. But note that what the programmer does is very different here. As an OOP designer, you are looking for elements to model, not procedures to decompose. ... A procedural program is like a tree of functions and an object program is like a web of clients and servers. These are very different of course, and to be an effective programmer of one kind or the other, you need to be comfortable with the basic job you are trying to do. The view of the nature of the computation (tree vs. web) needs to

> be natural to your thinking processes so that your typical thought processes lead you to one set of solutions rather than to the other.
> (Bergin 2000)

This has obvious implications for this project's objective of exploring the potential of patterns for the problem solving process. Clearly, Bergin's view is that the problem solving technique that students are learning during the first programming course is not reflecting the object-oriented nature of the course.

On the difficulties of learning to program there is a vast body of literature (Sleeman 1986) (Denning 1989) (Ayen & Grier 1983) (Maj et al 2000) (Bergin et al. 1999) (Harrison & Magel 1981), and because the observation of this phenomenon is ubiquitous, there are beginning to be many voices joining the "use of patterns in introductory programming" chorus. There are, naturally, many factors which impinge on this problem, such as a lack of any real interest in programming (Sheard & Hagan 1998) that educators can do little about. From our perspective, the main difficulty seems to be the non-acquisition of an ability to solve programming problems. Of course the teaching of problem solving skills is an issue that long predates computers and programming (Mayer n.d.) (Lockheed 1979), and there are many studies into the relationship between programming and problem solving ability (Mayer et al 1986) (VanLengen & Maddux 1990) (Reed 1998) (East & Wallingford 1997) (Haverty et al. 2000).

Most of the authors advocating the use of patterns in introductory programming courses seem to approach the idea from a cognitive perspective, and concentrate on the problem-solution pair aspect. For example, in the paper, "Patterns and Pedagogy", the authors attempt to assess the use of patterns in introductory programming courses from a knowledge integration perspective.

> Much psychological research ... suggests that programming expertise is partly represented by a knowledge base of pattern-like chunks, variously named plans, templates, schemas, or idioms. ... Components of such a chunk resemble those described in Design Patterns. ... Additional research suggests that students gain expertise in programming and other disciplines from a process of knowledge integration. Students integrate their knowledge by adding new pattern-like chunks, sorting out these chunks, creating new chunks, and restructuring promising chunks. ... Patterns can potentially help to organize the examples and code they see as they learn to program.
> (Clancy & Linn 1999)

The suggestion, here is that patterns are useful in the integration of knowledge, in the task of putting disparate concepts and ideas into a coherent, or more coherent form. Investigation of the cognitive processes involved in inductive reasoning has shown that the three fundamental areas of inductive activity are "Data Gathering, Pattern Finding, and Hypothesis Generation" (Haverty, Koedinger, Klahr & Alibali 2000). Inductive reasoning, the process of generalizing experience into concepts, is clearly an element of knowledge integration, and therefore patterns and their discovery are implicated as well. But the cognitive research that points to this connection between patterns and inductive reasoning, was based on studying the solving of inductive reasoning problems in Mathematics (Haverty et al. 2000). So it suggests a connection between patterns and problem solving, not just patterns and inductive reasoning as such. Nevertheless patterns are shown to be important in forming, or integrating knowledge in Clancy and Linn's terms, as well as in using knowledge in problem solving activities.

The paper goes on to address "the process by which patterns are inferred from examples and the mechanics for linking or connecting patterns." From their review of the research literature, the authors draw a number of conclusions.

> Novices don't infer patterns naturally (Linn).
> Instruction can't focus only on patterns (Mann).
> Expert patterns may be inaccessible to novices (Shackelford and Bake).
> Abstract understanding is needed for pattern application, as is a belief that reuse is appropriate (Hoadley).
> The context of a pattern's use in case studies can help (Linn and Clancy, Schank).
> Environmental support and control of the program design and implementation process can help (Hohmann et al.).
>
> (Clancy & Linn 1999)

These results are indicative of an important role for patterns in the acquiring of knowledge, as well of in developing problem solving skills, but, for our purpose, the most significant finding is the first in the list. This reinforces our own point about the misplaced reliance that educators tend to put on the principle of osmosis (see Section 1.5).

Some of the organizational issues involved in moving to patterns are discussed by Proulx, who also concentrates on cognitive issues (Proulx 2000). Harrison takes the organization line even further, suggesting "patterns for teaching patterns in a classroom setting". An interesting idea among these patterns is that students be taught "about the structure of a pattern by leading them through the process of writing a pattern as a group exercise" (Harrison 2001). If the course is about patterns specifically, this is probably appropriate, however we see patterns as a guide to programming, not as a subject in themselves. Another problem that we notice in many papers, is that they are still concentrating on patterns at too high a level for novices. For example, the Kaleidoscope project, advocated by Wick(Wick 2001) seems rather ambitious even if treated in a stage by stage manner. And, like the proposal by Astrachan for "Elementary Patterns" (Astrachan n.d.) it uses GoF level patterns, which are probably more appropriate in a second course.

Because of the obvious synergies between the pattern concept and problems this is the aspect emphasised by most advocates of the idea of using patterns in introductory programming courses. But if the indications of our own work in this project are correct, the obvious correspondences are not enough, problem solving is more than matching a problem with its solution, it's a *creative process* that derives from many such matchings, any complex problem cannot be solved by simple matching. Fortunately, these factors are not the only benefits offered by the pattern notion, full development of the language idea and the process idea being two that have been largely overlooked in the literature. Therefore, the fuller investigation of pattern theory from an introductory programming perspective that we are attempting in this dissertation promises to elucidate some of the less obvious benefits. Alexander's thinking revolves around patterns, language, and process, it is his contention that it is life (process) that gives form to the patterns we see all around us, not analysis based on logical and decompositional methodologies[1], and it is process (creativity) deriving from pattern language (conceptual order), that underlies the human condition.

[1]Analysis "may be a poor tool if used to prescribe the physical nature of forms, it can become a very powerful tool indeed if it is used to explore the conceptual order and pattern which a problem presents to its designer" (Alexander 1964, p. 7).

Chapter 3

The Mythos (Mystic Intelligibility) of Experience

Imagination is more important than knowledge.

(Albert Einstein, On Science (1954))

If we are to avoid the morass of metaphysics, we must reduce as many concepts as possible to numerical terms. On the other hand we must face the fact that the most important aspects of human life are intrinsically nonnumerical. Any attempt to ignore this is highly unscientific. In a true intellectual approach one accepts this fact and copes with it.

(R. Bellman 1962 (Massey) 1967, p. 67)

3.1 Creating the Mind out of Experience

Imagine a world in which nothing that happens bears any relation to that which has gone before, an existence where nothing is ever the same twice, where there are no recurrences, where experience is just a continuous stream of sense impressions flashing past. Well the point is that you can't. Not only because such an existence isn't possible - what is life if it is not a series of cycles, regularities and rituals; beating heart, breathing, waking-sleeping, drinking, eating and so on? - but also because it is impossible to imagine. Even the most fantastic of imaginings is infested with concepts like 'mirrors', 'rabbits', 'hatters', and 'little girls' called Alice, all of which must be familiar concepts before the fantasy can even begin. The doors of perception are opened not as Aldous Huxley contends by LSD, but only by the ghosts of happenings past. This is because imagination is an integral part of the development of mind, not the product of "random mind expansion", and it explains why one can't imagine what it is like to, for example, feel like a bat - one doesn't have the conceptual structure on which to base the imaginary experience, and no drug is going to alter that fact, because one doesn't possess the perceptual apparatus to develop it from real experience.

Imagination is fundamentally conceptual in nature, ‘to conceive’ is virtually synonymous with ‘to imagine’, after all - there is no ‘tabula rasa’. You can’t imagine anything that isn’t based on concepts that you already have in your head - “creativity is an automatic consequence of having the proper representation of concepts in a mind” (Hofstadter 1985, p. 528). Indeed, you can’t even imagine ‘nothing’. Nothing is no mind, another ‘tabula rasa’, and you need a mind to imagine anything, even ‘nothing’ - ‘nothing’ being a concept after all - no mind, no imagination. In the end it is concepts that we think *with* (Novak 1977, p. 18), not the brain, the point being that imagination is an important factor in human intelligence. Making decisions implies the need to assess the consequences of each of the possible options, and projecting one’s mind into the future requires imagination (Popper 1966, p. 233).

Maybe, just maybe, it is possible for the brain to be free of any conceptual structure within the first few minutes of being born, but even this is doubtful. The feeling for space, for time, for colour, for sensation itself - these are all built from recurring experience from the very moment that the sensations begin to flood in, so, at the very least, the potential for these concepts to form must exist in the brain. “Our senses are activated before we are born, and we begin accumulating experiences that structure our brains quite early. And ‘as the twig is bent ...’ according to the old expression, our competences develop” (Greenberg 2004*a*). If the flow of sensation was entirely random - as if, indeed there could be any such thing as random sensation - imagination would, in fact, could, never arise.

So the key to understanding, to the very soul of the way that the mind works, is the way that experience reveals the world in a process of unfolding potential. This unfolding of experience, a form of Alexander’s “fundamental process” by which the existing relationships between objects in the world transform themselves into the potentialities hidden as the seeds of the future in the current relationships, constitutes the dynamic “order of nature”, and accounts for the ordering power of mind.

> The natural world as a whole remains real for us, according to Husserl, precisely inasmuch as the process of the coincidence of predelineated aspects and actual subsequent perceptions continues. “The existence of a world is the correlate of certain experience-patterns marked out by certain essential formulations” [Husserl p.136]. The existence of each real thing and of the world as “the totality of objects that can be known through experience” [Husserl p.46] is thus presumptive or contingent in the sense that no complete fulfillment of the system of implicit references is possible. The “reality-status” of any and every physical object is always subject to modification in the light of subsequent experiences.
>
> Suppose now that in fact none of the “potential” aspects of current experience (aspects which, as we have seen, constitute the meaning of the perceptual object) were fulfilled from moment to moment. The

> world - the regular unrolling of ordered patterns - would exist no longer. ...
> Such a process would cease to make sense as the uncertainty about subsequent moments increased. Information-content would be maximal because it would be impossible to predict what would follow. Experience would be meaningless since "Reality and world ... are just the titles for certain valid unities of meaning [Husserl p.153], namely the Markoffian perceptual processes
>
> (Crosson 1967, pp. 126-7)

What we call "logical inference" is simply the extension of of "perceptual inference". In their state of existence in the world, mountains have aspects that we cannot perceive from any one particular viewpoint - the other side of the mountain. But it is only through experience that we learn the perceptual inferring of the other side, the perceptual order then is just a product of the unfolding experience of a mountain as we travel through various viewpoints. Its *actual* order, the structure of mountain, in the real world accounts for the conceptual order, "mountain", in our mind even though at any particular moment we cannot *actually* perceive the totality. The *wholeness* of the mountain we perceive, the very concept "mountain", is a product of the patterns of experience, not something that we can apprehend in a single moment. In other words the mind is ordered by the experience of order, the ability to *infer* unperceivable aspects of reality derives from two basic facts - the unperceived aspects are part of the structural order of reality, and conceptual order is as much a product of previous experience as it is of current perception.

In fact, the *conceptual order* of the mind consistently overrides those aspects of experience that don't make logical sense. That is, despite the experiential orderliness, there *are* many aspects of reality that, seem inherently 'weird' in terms of being conceptually inconsistent with other parts of reality. But we just accept them as normal without really considering how conceptually 'unusual' they are because this is, in fact, the power of the pattern, we don't have to think about it, or even notice its inherent 'weirdness', because it just keeps happening over and over again. Being a pattern means that it has lost the power to surprise us, in fact its power is the exact opposite, the power of normalcy. If you stop to think about the phenomenon of rain, for example, water falling out of the sky, this is quite 'weird' in a conceptual sense in terms of our other experiences of water. The point is that the fact that rain is, in most places, quite common, obscures this 'strangeness', and the fact that we accept it as quite 'normal' derives from its status as a pattern in our lives, not that it makes obvious sense. Water in its everyday manifestation in our lives simply does not float around in the air, `RAIN`[1] is in fact inconsistent with our other patterns concerning water, `LAKE`, `DAM`, `STREAM`, `SEA` ect., and one has to have a quite sophisticated understanding to 'make sense' of the phenomenon of rain at any level of logical analysis. But

[1]In this text pattern names will be indicated by appearing in this font - `RAIN`.

we don't analyse it, the basis of our ready acceptance derives from experience not analysis. The pattern phenomenon demonstrates that it is experience, not analysis, that gives rise to most of those concepts that are fundamental to our lives. Analysis, in fact, is dependent on concepts, not the reverse, just as synthesis is. I must have *some* concepts before I can analyse anything.

So the conceptual structure that develops in the human brain, beginning from day one - the mind - isn't a matter of pure chance, or even just an expression of the individual's genetic heritage, it is the result of the interplay between that heritage and experience, most of which is not only common to all of us, but consists largely of repeating form.

> We live in a universe of patterns.
> Every night the stars move in circles across the sky. The seasons cycle at yearly intervals. No two snowflakes are ever exactly the same, but they all have sixfold symmetry. Tigers and zebras are covered in patterns of stripes, leopards and hyenas are covered in patterns of spots. Intricate trains of waves march across the oceans; very similar trains of sand dunes march across the desert. Colored arcs of light adorn the sky in the form of rainbows, and a bright circular halo sometimes surrounds the moon on winter nights. Spherical drops of water fall from clouds.
> Human mind and culture have developed a formal system of thought for recognizing, classifying, and exploiting patterns. We call it mathematics. By using mathematics to organize and systematize our ideas about patterns, we have discovered a great secret: nature's patterns are not just there to be admired, they are vital clues to the rules that govern natural processes.
>
> (Stewart 1995, p. 1)

Anything that occurs in our mind is a part of a whole, that grand system that we know as the universe. As Kant pointed out, everything is either coexistent in time or follows from something that happened previously. "All substances, in the world of phenomena, in so far as they are coexistent, stand in complete community, that is, reciprocity one to another" (Kant 1881, p. 184). This "complete community" is both spatial and temporal in extent. "Every action, as a phenomenon, so far as it produces an event, is itself an event, presupposing another state, in which its cause can be discovered; and thus everything that happens is only a continuation of a series, and no beginning, happening by itself, is possible in it" (Kant 1881, p. 469). These two principles constitute the idea of "causality of nature" (Kant 1881, p. 460), often expressed as "the same events take place if the same conditions apply" (Weyl 1959, p. 192) and together with the "causality of freedom" (Kant 1881, p. 460), "a faculty of determination, independent of the necessitation through sensuous impulses" (Kant 1881, p. 461), form a dynamic community expressed in the "idea with regard to the totality of the derivation of cosmical events from their cause" (Kant 1881, p. 460), what we might call the 'human view' of the universe.

> The unity of the universe, in which all phenomena are supposed to be connected, is evidently a mere deduction of the quietly adopted principle of the communion of all substances as coexistent; for if they were isolated, they would not form parts of a whole, and if their connection (the reciprocity of the manifold) were not necessary for the sake of their coexistence, it would be impossible to use the latter, which is a purely ideal relation, as proof of the former, which is real. We have shown, however, that communion is really the ground of the possibility of empirical knowledge of coexistence, and that we can only conclude from this the existence of the former, as its condition.
>
> (Kant 1881, p. 190)

Whatever theory holds sway for the beginning of the universe what we do know is that it leads ultimately to this world in our head, to mind and all its properties. "Nothing can be born of nothing, for if it were otherwise, anything could be born at will from anything" (Yukawa 1973, p. 81). Like imagination, the universe cannot begin from a 'tabula rasa' because it ends up as something, indeed everything, even if it all turns out to exist only in our minds. Whatever the starting point was, it contained the initial conditions for the process, loosely called 'cosmic evolution.' Strictly speaking, of course, the process is not evolutionary in the modern sense because it does not involve a dynamic relationship of adaptation, it is simply a process of systematic change over time (Thorpe 1962, p. 2), that generates effects that recur time and time again. We know that much about our genesis. Universal natural laws are merely expressions of repetition, experience made predictable, and they must flow from the original conditions, from genesis. Everything, even the sense of self that is central to the human condition is made possible by the eruption into being that occurs at the very beginning of time. If this were not so then there would be no sense of self, no subject, to contemplate the universe, to think, to imagine and to dream. In some vital sense we are the mind of the universe - its imagination and its conscience. We know more than Descartes' 'I exist', we know 'I exist in contra-distinction to everything else.'

The fact that everything that we know comes ultimately from experience tells us that knowledge is mystic, that is, in principle, a mystery. This is not to say that it is 'magical' or 'supernatural' in any way, just buried deeply in the very structure of thought, at the interface of experience and behaviour. It is an aspect of *mythos* rather than *logos*, those features of the way that we live our lives that come under the heading, *spiritual*, in the sense that one lives one's life with *spirit*.

> Natural science knows nothing of the relation between behaviour and experience. The nature of this relation is mysterious - in Marcel's sense. That is to say, it is not an objective problem. There is no traditional logic to express it. There is no developed method of understanding its nature. But this relation is the copula of our science

> - if science means *a form of knowledge adequate to its subject.* [Emphasis in original].
>
> (Laing 1973, p. 16)

As human beings, our fundamental drive is to create understanding, and, as understanding one thing is a factor in coming to understand other things, the creative thread, buried deep in the almost instinctive side of our nature, is impossible to untangle - "creativeness is entirely irrational, a mystical faculty" (Popper 1966, p. 228) - just as the chain of cause and effect is impossible to unravel in a throw of the dice. To call an event random, or creative, is to state that it is unfathomable, mystic, but not because it is inherently indeterminate, just indeterminable in practice[2]. Just as the foundation of the individual mind and understanding is laid in the very early experience of the child, so the bases for our rational methods of reasoning, of creating understanding in the large, were established long ago and on the basis of the insight derived from those very same childhood experiences.

> To the Greeks we owe the insight that the structure of space, which manifests itself in the relations between spatial configurations and their mutual lawful dependences, is something entirely rational. Whereas in examining a real object we have to rely continually on our sense perception in order to bring to light ever new features, capable of description in concepts of vague extent only, the structure of space can be exhaustively characterized with the help of a few exact concepts and in a few statements, the axioms, in such a manner that all geometrical concepts can be defined in terms of those basic concepts and every true geometrical statement follows as a logical consequence from the axioms. Thereby geometry has become the prototype of a deductive science. And in view of this its character, mathematics is eminently interested in the methods by which concepts are defined in terms of others and statements are inferred from others. (Aristotelian logic, too, was essentially a product of abstraction from mathematics.)
>
> (Weyl 1959, p. 3)

So the defining characteristic of the human is creativity in the strongest sense possible, the human condition is itself a product of human creative potential, which is, in turn, an element of the human condition. Even our rational, scientific methodologies derive from the mystery of the creation of mind. The essential interaction between human and environment is creative in spirit, not deterministic.

[2]It was Poincaré who pointed out that we are limited always by our instruments. Measurements are always approximations no matter how precise. Reality is exact, it takes its measurements, in effect, to an infinite number of decimal places, not to a finite one like we necessarily do. Therefore the problem of determinism and the problem of predictability are distinct problems.

> For according to determinism, any theories - such as, say, determinism - are held because of a certain physical structure of the holder (perhaps of his brain). Accordingly we are deceiving ourselves (and are physically so determined as to deceive ourselves) whenever we believe that there are such things as arguments or reasons which make us accept determinism. Or, in other words, physical determinism is a theory which, if it is true, is not arguable, since it must explain all our reactions, including what appear to us as beliefs based on arguments, as due to *purely physical conditions*. ... This means that if we believe that we have accepted a theory like determinism because we were swayed by the logical force of certain arguments, then we are deceiving ourselves, according to physical determinism; or more precisely, we are in a physical condition which determines us to deceive ourselves. (Emphasis in original.)
>
> (Popper 1972, pp. 223-4)

3.2 The Pattern of Experience

As conscious purposeful beings we stand on ground that we may never be able to precisely delimit. Yet this should not worry us because we do things every day, which, but for the fact that we accomplish them every day, should astound and amaze that. Think of unexpectedly meeting someone who you once knew well after an absence of twenty or thirty years. Recognition is done unconsciously with complete certitude.[3] Any momentary hesitation is only due to the conscious realisation that this feat should be impossible. To appreciate its impossibility, attempt to explain in rational terms how this recognition process works, how you succeed “in separating essential from unessential features” (Weyl 1959, p. 286) given that many changes have occurred in the appearance of your acquaintance in the meantime. The foundations of mind, set in the time before language had gathered its communicative power, in the time before the *mythos* had become the *logos*, before words had taken over from raw meanings and feelings as the building blocks of consciousness, must remain ever mysterious. Here even memory is but

[3]Two of the most striking capabilities of human memory are those pointed out by Pribram. “The first is our ability to recognize a person we know, when he appears in our field of view, which may contain a hundred more people. The sudden flash of recognition we may feel, this absolute certainty of ”this is him and it can be nobody else”, is not just a subjective emotion, but is apparently evoked only by an extremely reliable and fast form of information processing in our brain. This function of recognizing is also performed by the two-dimensional hologram, as the appearance of a bright light point in the image plane of the optical arrangement, and the brightness and sharpness of the light point are a scientific measure of the degree of recognition.

The second capability is our ability, after recognizing a person, to recall quickly a considerable amount of the information we have about this person. In an optical arrangement, the recognition signal given by the twodimensional hologram provides the instruction for generating total recall of the relevant information from a three-dimensional hologram (Pribram 1971, p. 156) the raw engram, the 'neuronal-association-level'[4] recording of direct experience not yet organised into a conceptual model of the past, and therefore hardly worthy of the designation. Yet this undifferentiated and largely unstructured mist of experience provides the material out of which both the adult and the modern mind emerge.

No wonder that we feel that much of what we do and who we are is intuitive and instinctive, even virtually innate. We live in a sort of 'twilight zone' between 'making sense' and 'sensing', between conceiving and perceiving, where the connection between ideas and reality is fundamentally obscure. But this essential obscurity does not diminish its fundamental role in making us what we are. The idea of 'human nature' has largely fallen from favour, or at least mention, in recent times, and mostly for good reason. But consider the vast experiment in delineating 'human nature' that was conducted between neolithic times and 1492. From the time that rising sea levels separated America from Europe, neolithic humankind began two lines of cultural development in isolation from each other. At the time of separation these societies were village-based systems that had developed out of the transition from the hunter-gatherer to early farming lifestyles. Yet when these two separately developed cultures came into contact with each other again, during the 15th and 16th Centuries, each was able to recognise essentially identical forms of social and cultural organisation in the other. Although different in detail, the elements of social, political, religious and intellectual life all took the same basic shape, and formed a similar 'social landscape' in both the European and Aztec 'civilizations'.

This tells us about the power of the basic human pattern language in shaping human cultural development. 'Human nature' is just that dynamic interaction between the human biological form and the environment. Everywhere we face the same basic conditions, have the same needs and problems. It should not be surprising that we develop the same 'mind' in response, that we operate with the same basic pattern language. We are essentially animals, our intelligence is a product of evolution, just as any other attribute of an animal is, and the basic difference between plant and animal life is behaviour - the ability and the need to make decisions about what behaviour to perform, what 'to do' in a given situation - derived from the possession of a system of coordination based on electrical impulses (nervous system) as well as the much slower chemical-based (hormonal system) dynamics inherited from plant life.

So almost as soon as you have behaviour you have a requirement for learning because it is a dynamic attribute. Animals 'learn' by observing the patterns of behaviour of their elders, and it is from this behavioural pattern language that

[4] "The unit of analysis for brain function has classically been the neuron. The present proposal for a two-process mechanism recognizes an additional unit: the neural junction, whose activity can become part of an organization (the slow potential microstructure) temporarily unrelated to the receptive field of any single neuron. Neural junctions are thus much more than just way stations in the transmission of nerve impulses" (Pribram 1971, p. 25).

our mental forms develop. Human behaviour is fundamentally the ability to make decisions and there is really only one way to do this, to build on what you know. In the end "people make decisions based on their recognition of similarities between past experiences and current situations" (Hopkins & DuBois 2005).

The primary fact about being human is the drive to create mind from experience, so it is, in principle, impossible to separate this motivation to know from knowledge itself. What we call 'mind' is a continuous ongoing process, a work-in-progress, never a completed stand-alone artefact. We learn to trust the regularities of experience, the order in what we perceive, and from this we build the ability to organize perception, to conceptualize. The initial state in which we find ourselves is cognition without the cognitive power of narrative and understanding, we are sensing without 'making sense'. Yet, like a story in which some vital information is withheld until close to the end, we are building the basis on which, eventually, sense will be made. 'Making sense' of our experience is the ultimate creative act, "the art of understanding" in Heidegger's sense (Heidegger 1962, p. 194), and this is why the "*creative spark* is not the exclusive property of just a few rare individuals down the centuries, but quite to the contrary, it is an intrinsic ingredient of the everyday mental activity of everyone" (Hofstadter 1985, p. 527).

The true power of the metaphor underlying cognitive science, the brain as a computer, is not that it is true in any metaphysical sense, just that it provides a way of thinking about what goes on inside the head[5], so as the researchers at the Sony Computer Science Laboratory (SCSL) in Paris say, the aim of research like their own should be, not to build artificial intelligence, but to help "understand how children learn" (Frederic Kaplan quoted in IDG News Service (14th June 2005)). Because, as Wilder Penfield, one of the pioneers of neurological research, says, there is "no suggestion of action by a brain-mechanism that accounts for mind-action" (Penfield 1975), this is the ultimate 'black box' - one can never deal directly with the brain-mind interface. It is not even clear that psychology is reducible to neurology let alone to digital logic.

> If psychology is reducible to neurology, then for every psychological kind predicate there is coextensive neurological kind predicate, and the generalization which states this coextension is a law. ... [But] there are no firm data for any but the grossest correspondence between types of psychological states and types of neurological states. ... The present point is that the reductionist program in psychology is clearly *not* to be defended on ontological grounds. Even if (token) psychological events are (token) neurological events, it does not fol-

[5]This is not to imply that people *can't* be 'programmed', that, after all, is the very *purpose* of propaganda (the word 'propaganda' coming originally from from Mediaeval Latin for 'propagating the faith'). But we tend not to associate purely 'programmed' behaviour with intelligence when it is exhibited by humans so it is somewhat surprising that we can imagine attributing intelligence to a 'programmable machine'.

> low that the kind predicates of psychology are coextensive with the kind predicates of any other discipline (including physics). That is, the assumption that every psychological event is a physical event does not guarantee that physics (or, a fortiori, any other discipline more general than psychology) can provide an appropriate vocabulary for psychological theories. I emphasize this point because I am convinced that the make-or-break commitment of many physiological psychologists to the reductionist program stems precisely from having confused that program with (token) physicalism.
>
> (Fodor 1975, p. 17)

The point is that every computational state, by definition, depends on prior computations, on a computational cause, but we know, as an established empirical fact that mental states do not necessarily have purely mental causes. A radical state of ennui can be due to the state of the oysters that one ate yesterday rather than resulting from general dissatisfaction with the state of the world. So, at least "some mental states are, as it were, the consequence of brute incursions from the physiological level" (Fodor 1975, p. 200), and there are even suggestions, admittedly controversial (Khamsi 2004), that stimulating the temporal lobe of the brain can induce feelings of religious fervour (Ramachandran & Blakeslee 1999, p. 175) and out-of-body type feelings (Blackmore 1994, p. 29), suggesting that it is not even true that every mental state can be defined by mental relations, let alone computational ones.

The similarities in function between the brain and computer enable us to draw some cautious parallels (Winograd 1983, p. 13), provided that the clear differences are also kept in mind. So, for instance, the brain as computer idea reduces the brain's function to cognition. But it is clear that "as humans, we experience the world in aesthetic, affective, and emotional terms as well" (Winograd 1996, p. xix) as cognitively. We process experience in all its facets, not just as information. Moreover, "whereas computers still perform calculations in a linear order, the human brain can make a continuous series of computations at the same time, passing information back and forth in a non-linear, self-organizing manner" (Chabria 2005).

But the fundamental difference lies in that property of experience that we call *meaning* that derives from biological 'being'. As we discuss further in Chapter 5, the case of the young deaf and blind girl, Helen Keller, demonstrates the essential 'emptiness' of her symbol for 'water' before the moment of revelation when she discovers *meaning*. "I knew then that w-a-t-e-r *meant* [emphasis added] the wonderful cool something that was flowing over my hand. That living word awakened my soul, gave it light, hope, joy, set it free!" (Helen Keller quoted in (Langer) 1976, pp. 62-3) . Before her moment of enlightenment Helen Keller was processing signs, not meanings or even symbols, she was *computing* not *thinking* because she did not have the web of associations that normally sense endowed people acquire through direct experience. For there is a logical fallacy involved in the notion of a computer processing symbols as Fodor points out.

> Following Turing, I've introduced the notion of computation by reference to such semantic notions as content and representation; a computation is some kind of content-respecting causal relation among symbols. However, this order of explication is OK *only if the notion of a symbol doesn't itself presuppose the notion of a computation.* In particular, it's OK only if you don't need the notion of a computation to explain what it is for something to have semantic properties.
> ... Suppose, however, it's your metaphysical view that the semantic properties of a mental representation depend, wholly or in part, upon the computational relations that it enters into; hence that the notion of computation is *prior* to the notion of a symbol. You will then need some *other* way of saying what it is for a causal relation among mental representations to *be* a computation; *some way that does not presuppose such notions as symbol and content.* It may be possible to find such a notion of computation, but I don't know where. (Certainly not in Turing, who simply takes it for granted that the expressions that computing machines crunch are *symbols*; e.g. that they denote numbers, functions, and the like.) The attempts I've seen invariably end up suggesting (or proclaiming) that *every* causal process is a kind of computation, thereby trivializing Turing's nice idea that *thought* is. [Emphases in original].
>
> (Fodor 1998, pp. 11-12)

The computer processes 'empty' symbols, mere signs, and the mind deals with *meanings*, the web of associations acquired through experience of the actual entity to which the sign acts as symbol. Although there *is* an association between the two it is not the case that the symbol is the *same thing* as its meaning, it is an *essentially arbitrary*, that is, conventional, not experiential, representation of the meaning, and this is the difference between a formal definition and a pattern, or a pure abstraction and a real entity. Life is a continuous dynamic process in a physically continuous 'space' out of which *meaning* emerges, whereas the *expression* of meaning relies on the use of a system of discrete symbols, and these are fundamentally different processes occurring in fundamentally different 'worlds'. If the mind is a 'machine' at all, then it is a *mythos machine*, while the computer is a *logos machine.* More properly, of course, mythos implies organism, not organisation.

We consider it plausible to argue that there is a fundamental disjunct between definition (logos) and meaning (mythos) and that the pattern concept provides the means to cross the gap, because pattern is an expression of some aspect of real world process in terms of at least some of the associations that form the web of meaning of direct experience, the 'context' in pattern terminology. Indeed, in an attempt to escape the circularity of the idea that concepts are definitions, cognitive science has spent the last decade defining concepts as *prototypes* on the statistical basis that "everybody who has a concept is highly likely to have its prototype as well" (Fodor 1998, p. 93). That is, "ask a subject to tell you the first — that comes into his head, and it's good odds he'll report the prototype for the category —: cars for vehicles, red for colours, diamonds for jewels, sparrows for birds, and so on, Ask which vehicle-word a child is likely to learn first, and prototypicality is a better predictor than even very good predictors like the relative frequency of a word in the adult corpus" (Fodor 1998, p. 93). It seems to us that substituting the idea of a pattern for 'prototype' retains the circularity-escaping aspect while avoiding the problem of compositionality.

> In a nutshell, the trouble with prototypes is this. Concepts are productive and systematic. Since compositionality is what explains systematicity and productivity, it must be that concepts are compositional. But it's as certain as anything ever gets in cognitive science that prototypes don't compose. So it's as certain as anything ever gets in cognitive science that concepts can't *be* prototypes and that the glue that holds concepts together can't be statistical. [Emphases in original].
>
> (Fodor 1998, p. 94)

Patterns have more combinatorial potential than prototypes precisely because of what they are - they *occur* during the functioning of a system, that is, dynamically, and they form a language. So, the person who 'has' the appropriate patterns from experience will automatically *have* the concept in a way that the person who tries to acquire it through a dictionary definition will not. Moreover the compositionality of patterns cannot be doubted because most, if not all, concepts, are themselves made up of several patterns. Just think of all the patterns from the various senses that make up the concept 'water' that were missing from the young Helen Keller's universe.

So there would seem to be a qualitative difference between the *meaning* that arises from real life experience and the operational semantics of a 'symbol' processing system, because, in actual fact, operational semantics is nothing more than logical syntax.

> For the user, the computer function can be operationally described as a physics-free machine, or alternatively as a symbolically controlled, rule-based (syntactic) machine. Its behavior is usually interpreted as manipulating meaningful symbols, but that is another issue. The computer is a prime example of how the apparently physics-free function or manipulation of memory-based discrete symbol systems can easily give the illusion of strict isolation from physical dynamics.
>
> (Pattee 2001*b*)

It is this "physics-free", that is, "real-world-context-free", aspect of computing systems that is the both the source of their power, and the font of the difficulties humans have in using, and especially, programming, them. One of the major influences on the development of modern software is the quest to make it more suitable for human consumption, so to speak.

This is because the computer requires a strictly non-negotiable context for the symbols it processes, made up of a strictly defined operational 'meaning', if it can even be called that, for symbols and the logical syntax for their manipulation, whereas the mind seems not to be syntax driven at all. Rather it is almost entirely context-flexible; what matters for the use of a symbol is semantic context, not the rigid syntactic one, that's why it is a pattern, not simply a symbol - it carries a web of associations not just a dictionary definition. Mostly we don't need a definition to 'know' what a word means, we have derived its meaning from experience **which is *always* contextual, *never* abstract**. The word, in this case, is essentially a pattern because it is embedded in the web of experience that constitutes our personal pattern language, and it is this factor that explains why we often find it difficult to expound upon the meaning of a word, we live its meaning, we don't define it - meaning is dynamic whereas definition is static. Definition and syntax only become important in the act of communication between minds, and even then it is infinitely less formal than a programming logic needs to be because it needs to impart meaning not to formally establish a conclusion from fundamental principles.

But there is a fascinating contrast here, caused by the fact that we have a tendency to see what the computer does as 'solving problems' as if it were an active participant in the design of the solution, as if it were 'thinking', as if it were 'creative'. The point is that problems exist *only* in human terms, no computer ever *had a problem* in the way that humans have them. Even if it fails to switch on, fails to boot as we say, this is a problem for the human, not the computer, and likewise the solution has to be discovered by the human - problems and their solution exist only in *human conceptual space.* What the analogy overlooks is that this is a logical system. Logic cannot create, it can only be *used* to *discover* solutions that are inherent in the situation that is causing the problem, to deduce them from first premises. As Alexander explains, physical form can only be logically determined from original state to the extent that the original state requires it, and this is true for conceptual form as well.

> Logic, like mathematics, is regarded by many designers with suspicion. Much of it is based on various superstitions about the kind of force logic has in telling us what to do. First of all, the word "logic" has some currency among designers as a reference to a particularly unpleasing and functionally unprofitable kind of formalism. The so-called logic of Jacques Frangois Blondel or Vignola, for instance, referred to rules according to which the elements of architectural style could be combined. As rules they may be logical. But this gives them no special force unless there is also a legitimate relation between the system of logic and the needs and forces we accept in the real world. Again, the cold visual "logic" of the steel-skeleton office

> building seems horribly constrained, and if we take it seriously as an intimation of what logic is likely to do, it "is certain to frighten us away from analytical methods." But no one shape can any more be a consequence of the use of logic than any other, and it is nonsense to blame rigid physical form on the rigidity of logic. It is not possible to set up premises, trace through a series of deductions, and arrive at a form which is logically determined by the premises, unless the premises already have the seeds of a particular plastic emphasis built into them. There is no legitimate sense in which deductive logic can prescribe physical form for us.
>
> (Alexander 1964, pp. 7–8)

And so it is with the computer. The machine provides the logic, the system of rules by which it operates. So there is no sense in which it can derive solutions that are not inherent in the premises of the logic. It runs *our* solutions - solutions to problems that exist outside of the strict limits of machine logic. Just as there is "no legitimate sense in which deductive logic can prescribe physical form for us", there is no way that a computer can itself solve a real world problem. The statement that Turing's "results entail ... that a standard digital computer, given only the right program, a large enough memory and sufficient time, can ... display any systematic pattern of responses to the environment whatsoever" (Churchland & Churchland 1990, p. 26) is not a correct interpretation of the Church-Turing thesis. Even in purely logical terms there are functions that cannot be "solved" on a Turing machine.

> Any device or organ whose internal processes can be described completely by means of effectively calculable functions can be simulated exactly by a Turing machine program (provided that the input into the device or organ is itself Turing-machine-computable, which is to say, is either finite or expressible as a computable number, in Turing's sense ... ; but any device or organ whose mathematical description involves functions that are not effectively calculable cannot be so simulated. As Turing showed, there are uncountably many such functions. (Examples from logic are Turing's famous halting function ... and the function D whose domain is the set of well-formed formulae of the predicate calculus and whose values, D(x), are 1 or 0 according to whether x is, or is not, derivable from the Bernays-Hilbert-Ackermann axioms for predicate logic.) It is an open question whether a completed neuroscience will employ functions that are not effectively calculable.
>
> (Copeland 2002)

Logic, like mathematics, is a means to an end, not the 'space' in which the end exists. The end, the solution, can only exist in the 'problem space', not the 'logic space' of the machine, so it cannot be derived by logic alone. Design problems are 'human problems', 'human ends', they require 'human solutions'. Reasoning is how we solve problems and how we convince ourselves. Convincing someone requires more than logic it requires reasoning, an appeal to the whole person, the mind, not just the logical or computational aspect of their brain. The holistic nature of the working of the mind is most evident in the way that we ‘appreciate’ a cultural experience such as a film. Most films are totally unconvincing at any level of logical analysis, yet one can be swept along, ‘convinced’ at the emotional level, to such an extent the that the logical inconsistencies and even the rational absurdities are ignored. Clearly, the mind is prepared to “suspend disbelief” as we say, if it is engaged at some level of mental activity, and this is important for the creative impulse.

Education can be regarded, albeit simplistically, as a form of persuasion, one is trying to ‘convince’ people about a ‘discipline’, a way of thinking about some aspect of existence. Logic is important in this regard, but it is not all that is happening in learning. Even the idea of ‘cognition’ where it stands “for any kind of mental operation or structure that can be studied in precise terms” (Lakoff & Nunez 2000) does not cover the case as is shown by the attempt to set up a taxonomy of educational objectives where examination of the cognitive domain was followed by an examination of the affective domain. Even Benjamin Bloom, the organiser of the report on the cognitive domain, recognised that “education as a process was an effort to realize human potential, indeed, even more, it was an effort designed to make potential possible” (Eisner 2000). Creating potential is, more or less, what we mean here by creating meaning or ‘mind’, and it encompasses more than those mental operations that can be “studied in precise terms”, it is, in fact, a feature of all aspects of life whether or not they can be precisely delineated. If this were not so then we would surely not still be casting around in the attempt to find ways to teach things as ‘limited’, as ‘precise’, and as ‘logical’ as instructing a machine.

3.3 Programming is just Thinking

But this is the root of the problem, the very precision and logicality of the machine. The basic cause of the problem is that the student of programming is being forced to deal with a system of strict logic and this is just *not* the normal way in which humans think. We are not good at handling strict logic, and this shows up in other areas where one has to deal with systems of this kind. One sees the same sort of difficulties when one is teaching basic unix, and even mathematics. It is an interesting historical fact that the way that the difficulties that novices have in dealing with a command line system of the unix type was to hide the logic behind a language of visual symbols. The modern operating system of a computer is a metaphor for office life with ‘desktops’, ‘files’, ‘folders’, ‘text pads’, ‘archiving’, ‘scheduling’, ‘mailers’, ‘memos’, ‘rubbish bins’ and the like. Even the internet had to come to be presented in a form, the web, more like the familiar directory style services, a sort of truly ‘universal’ business directory, before it achieved widespread acceptance. These metaphors all act to disguise the digital and logical reality of the basic computer system, so that, for example, when you place a file in the rubbish bin you are really deleting a string of binary digits from the storage medium, usually a hard drive[6], of the computer, and when you send a message on the email system you are generating ‘packets’ of such digits to be sent over the telephone system.

But the really interesting aspect of all this from our point of view is that the visual metaphor is based on the familiar patterns of life in an office, this is essentially a graphical representation of the *pattern language of the office*, the purpose of which is to hide from unfamiliar minds the rigid realities of binary arithmetic and symbolic logic. Like all metaphors, this one is a way of making the unfamiliar seem familiar in order to promote understanding, to enable us to deal with it using our normal modes of thinking, the patterns of mind that we have developed out of normal everyday experience. That this should take the form of a pattern language should not therefore be surprising, because, on its own, metaphor imports meaning, and an operating system requires more than static definition it requires ‘operational understanding’. What the graphical operating system really imports is the ‘operational semantics’ of the office because the real life ‘meaning’ of binary symbols and logic is non-existent for most people, it is not a pattern of everyday human experience, whereas the office experience, to some extent at least, is.

Clearly the difficulties we face in teaching programming are of the same kind, there is no ‘operational semantic’ inherent in making a machine perform a task to our requirements that derives from the patterns of everyday life. The closest analogy we can think of is natural language, it has the same sort of creative combinatorial potential, the trouble is that the communicative relationship is different. Natural language, as a means of communication, is based on both nodes of the interaction being conscious beings capable of flexible interpretation of the symbolic system in use, on apprehending what is *meant* rather than what is actually *said*. However this is not true of the programming situation; here, one of the entities involved is a rigid system of logic, it has no means to ‘interpret’ what is being said, to apprehend what is actually *meant* from its own experience as a human does. This is, in fact, the source of bugs in programming. A bug is simply a manifestation of the inability of a computer to infer meaning from context as humans constantly do, and it is a direct result of seeing the machine as a ‘communicative’ rather than a purely ‘computational’ entity.

So programming is, if it can be called by that name even, ‘communication’ of a restricted kind. In a sense, the computer has no pattern language, just a

[6]Of course, even this level of understanding turns out to be fairly metaphorical in practice as the sequence of binary digits carrying the information is not actually magnetically erased, just dereferenced in the space allocation (indexing) system of the storage medium, and is still accessible to a person with digital level nous. So a naïve user can actually be misled by the modern operating system if the idea of putting a file in the rubbish bin is taken too literally - as equivalent to shredding a paper file, for example.

system of symbolic logic, precisely because it *has no context in real life*, the only place where meaning exists. The 'humanness' of a person derives from life, from the fact that this is an organism set in a context - all meaning derives from this contextual relationship, so if a computer is to be considered in any way analogous to a human, that is, as a sort of 'being', then it is a peculiarly *context-free* being. Its only 'context' is symbolic logic. This makes 'communicating' with it a total nightmare for the human inexperienced in logic, but 'driven' by the pattern language of everyday life. One can't rely on the machine 'knowing' anything, of having the patterns of everyday thinking of a human. Its 'operational semantics' is that of an automaton, not a being. Even when implementing an algorithm, a human will apply the instructions creatively, varying a recipe according to personal taste, for example.

The move, then, to make programming languages the *same* as natural language, founders on the shoals of reality - it is simply impossible. Communicating with a machine is a problem of a different order to human communication. The thrust of the *underlying* idea is correct, we *do* need to be using the same sort of thinking in devising a program that we use in real life, because that's the sort of thinking that we are 'naturally' good at - we've spent our lives getting good at it - but we can't use it in *communicating* with the machine. Ergo we have to separate the thinking about a program, the design, from the medium of communication with the computer, the programming language. "The failure of information systems stems not from activities during their implementation but usually because of poor planning in the early stages of systems development" (Kaiser 1985, p. 2), and the poor planning comes about because, in using programming languages, we are forced to concentrate on the contingencies of the formal system rather than whatever it is that we are trying to build.

The need to separate design and implementation has been long recognized but the implementation of the separation, software engineering, was flawed from the start. It separates program design from the logic matrix, certainly, but engineering is the wrong metaphor, firstly because engineering techniques are almost as unfamiliar to most of us as is the system that we are trying to hide, and, more importantly, because it is almost as formal and logical. You don't implement the sort of free-thinking creativity that people display in everyday life by imposing engineering principles on them. Engineering is an appropriate metaphor for large scale programming projects where one can rely on the already developed creative skill of an *experienced* programming team, but it adds nothing to the process of acquiring the skill.

And it is in this context, acquiring the skill, that the difficulties caused by the strict logical and syntactic rigidity of programming languages really come home to roost. The model for any language acquisition task is, or should be, the method by which we learn to speak. So although grammar can be seen as a set of formal syntactic rules that underlie the use of language, we *absolutely do not* learn to speak by learning the rules and applying them. Rather we learn to speak with no regard to the formal system at all, in fact we are totally unconscious of it until, at a much later stage, our English teachers use it as an instrument of torture (Porter et al 2005, p. 237). So the model that we mostly use to teach programming is the *instrument of torture* model, not the natural language acquisition model. We teach the programming process as the use of a formal syntax, not a means of expressing ideas. Ideas drive expression, not syntax. The formal rules are the structure involved in *organising communication*, the ‘scaffolding’ in which the organising process takes place, not the force that does the organising.

In this regard the move to Object Oriented programming can be seen as a move in the right direction, but it, too, founders on the fact of it being implemented in programming language form. It recognises that we need to use familiar concepts from our normal experience in thinking about a program, but it still ties the thinking process to the communication process. In a sense it implements Aristotlean logic rather than machine logic (Rayside & Campbell 2000), but this just changes the notational form - in operational terms it is just as strictly formal and deductive. Everyday thinking is based on the objects and events that occur in the real world, but we deal with them as patterns of life rather than logical formalisms - deduction is an element of reasoning, not its totality. What the object oriented approach failed to do was to separate thinking from coding, and what you are left with when you do actually make the separation is object oriented thinking, that is, pattern thinking, not object oriented language.

So it is no coincidence that Alexander’s idea surfaced in object oriented programming circles. Look at Alexander’s pattern language - what you see are the objects and events of the built environment presented in a form that makes combining them a matter of normal style thinking rather than formal architectural training. This is a *language* for building, not an architectural prescription, it derives from everyday experience, not training. It recognises that creativity is a function of thinking, not of applying architectural or engineering principles. You need to have an idea first, before you begin thinking in terms of implementing it in architectural form, or in any real world form, for that matter.

The transformation of any field of human endeavour into a ‘discipline’ to be studied in the abstract has something of this tendency to formalisation of technique, to ‘discipline’ thinking about it. Discipline is certainly required in thinking about a subject of any difficulty, but taken to extremes, as is the case with logic, it must tend to stifle free ranging thought, and therefore, creativity, at least until such time as the system of logic has been comprehensively internalised. So the basic premise proposed here, ‘programming as thinking’, is essentially that of Peter Naur’s “programming as theory building” (Cockburn 2000) as most of what serious thinking is about is building an understanding of the world, that is, constructing or modifying a theory about some aspect of experience.

3.4 Learning to Program

The basic thesis of this dissertation is that the ability to learn a skill like computer programming derives from the fact of our existence as human beings, and that, therefore, it is, in principle, just an extension of our problem solving capabilities (Bergin et al 1997, p. vii). Yet, in practice, it would seem from the experience of educators in this field that many apparently 'intelligent' students find learning to program a difficult and even traumatic experience, raising the question about why this should be so. Is it something about the activity of programming itself that makes it apparently so difficult to learn, or is it a failure in teaching practice? On the answers to these questions turns the response to the question 'what is to be done?' We attempt to demonstrate here that the main source of the difficulties lies in the pedagogical approach to programming and suggest that the way to address the problem is to bring it closer to the way that we learn to speak. This is not to suggest that teaching any activity that requires problems to be solved is easy. We recognise, indeed, that all creative endeavour is difficult to teach. So our basic premise is that while creativity is massively difficult to teach, it is, or at least on the evidence of the ease with which we learn to speak, it should be, easy to learn.

Going by the struggle that many endure, it is clear that to the beginner the programming domain must appear to be like the world that we discussed at the beginning of this chapter, a world in which everything that happens is new, where nothing ever repeats, where total confusion reigns and where imagination is frozen into inactivity. So the question is what is causing the experience to be so disorienting?

> People commonly start their programming career by learning a programming language, usually through an introductory course. Inevitably, in the early stages their dominant concern is with mastering the language constructs and the fundamentals of breaking down and reorganizing a problem into a form to which those constructs can be applied. Some introductory courses concentrate so much on the language and how to use it to express the solutions to problems, that virtually everything else is excluded, though the better ones do stress principles, such as structured programming techniques, which are not language specific or problem specific. It is usually only later, when one has started programming seriously (either through becoming a professional programmer, or because programming is an important adjunct to other work) that the realization gradually dawns that there is a good deal more to being a programmer (whether a professional or a serious amateur) than simply coding. Indeed, for some the realization seems never to dawn at all.
>
> (Meek et al 1983, p. 12)

The authors' point here is that an introductory programming course is, most often, an introduction to a programming language, *not* programming. Could it be that this early concentration on the language, and therefore, code, is *not* the right way to introduce programming?

To illuminate this proposition we examine the various factors involved - the activities of programming, of learning, of creating, of thinking - to find what the differences and similarities are. For although programming encompasses much that is amongst the most complex tasks that we undertake, at its most basic level, it really is fairly simple.

> Given the diversity of tasks that computers can do today, people naturally find it very surprising that the computer's built-in abilities are so primitive. When a computer comes off the assembly line it will be able to do only arithmetic and logical operations, input and output, and some "control" functions. These capabilities comprise the *machine language* of the computer.
>
> (Horowitz 1983, p. 1)

So if the number of operations that a machine can perform is actually extremely small, then there has to be an explanation, other than machine language complexity, for the difficulty of programming. The difficulty, it would seem, arises mainly from the sheer audacity of the activities that we attempt to automate - the complexity is a function of the system being produced not the programming environment being used to produce it, or, as we try to argue here, it doesn't have to be.

> Many authors have convincingly demonstrated that computer software belongs among the most complex products of human endeavour, chiefly because it encompasses phenomena covering a very wide spectrum of specific times. For example, a banking house software is influenced by phenomena whose specific time is measured in years (loans, mortgages) and in microseconds (conflicts of concurrent accesses to a particular record field), which gives a span of some 13 - 14 orders of magnitude. Only when investigating natural objects, such as the Universe or the human body, has mankind ever encountered so complex systems, and never before has it constructed one.
>
> (Turski 1979, p. 450)

So programming is like composing music - the tool, musical notation or machine instruction set, is, inter se, quite simple. It is the ramifications of the combinatorial process, of creativity, that introduce the complexity we see in both programs and musical pieces. The notion that some entity can be 'inherently' complex implies that complexity is something that exists in its own right, rather than being a result of the entity's composition. Complexity is nothing more than the result of combining many smaller, less complex, and, at base level, simple elements, it is little more than an expression of multiplicity, indeed creativity. In fact complexity is just order, "hidden order" (Holland 1996) one might say, and it is simply the result of some process that is creating order. The real source of complexity in systems can be discerned in the remarkably similar case of DNA. Here we have a code consisting, like the computer's base operations, of very few fundamental elements (four to be exact), yet in its expression we see the truly amazing diversity of form that we call life, showing that it is the creative impulse, the 'purpose' if you like, of the code system rather than the code system itself that gives rise to the complexity of life on Earth. Complexity in software is the result of the creative expression of our need for order, that is, what we desire the machine to do, rather than any 'native' complexity of programming or programming systems.

But creativity is the mark of the human, we spend our lives 'creating' meaning, or understanding, uncovering the "hidden order" in the world around us. Most of what is most distinctively human is 'creative' at least in terms of the individual, if not the collective whole of humanity. Johnson-Laird has framed a working definition that covers "both highly original productions and the imaginative thoughts of daily life" (Johnson-Laird 1993, p. 117) where an act of creation is defined as a process that yields a result with the following properties.

> 1. The result is formed from existing elements, but in a combination that is novel for the individual and perhaps for society as a whole. It is not merely perceived or remembered.
> 2. It satisfies pre-existing criteria.
> 3. It is not constructed by rote or derived by some other simple deterministic procedure. The process allows for freedom of choice.
>
> (Johnson-Laird 1993, p. 117)

A computer program is clearly "novel for the individual" who produces it even when it addresses a previously solved problem and producing it is therefore an act of creation at the individual level. But if creativity is the mark of the human then something about the mix of programming and creativity is not working. If programming is an expression of creativity and creativity is our principle characteristic, we should, on the face of it, find learning to program at the basic level easy. No educator expects a novice to write a compiler (or a symphony) first up, the complexity of novice-level tasks is deliberately set low. Yet even this level of task is beyond many beginners, which suggests that something about programming, or the way we teach it, is stifling the native creativity that humans express in their everyday existence. Mind is an "aggregate of ideas" (Bateson 1973, p. 21) organised as a system for the purpose of contributing to the maintenance of the organism of which it is a part. It's difficult to see how we could have such a construct at birth - and that's the point. As a construct it has been created out of the raw perceptions of human experience. The mind is self-generated, both the product and the source of human creativity.

So the activity of teaching programming would seem to overwhelm the native capacity of the mind to create, suggesting that either programming itself, or the teaching effort, is structured in a way that does not address the way that the mind creates. Given that creativity is a process of organising information in novel ways, we should look to the information processing aspect. One of the main tasks of the mind is to structure the information that the organism has about the world in such a way as to facilitate the incorporation of new information as this is the only way that it can adapt to changing circumstances. Even narrative is important in this sense in that it endows a sequence of events with a significance, an ordering principle, that the simple sequence does not itself possess (White 1987, p. 14). The brain is a limited resource and would be severely overloaded if every experience had to be dealt with in a serial fashion, the system would simply break down. Structuring the information is important, then, not only in terms of storage capacity, but also in terms of retrieving it when needed. In this case most of structure is about context and significance, so mind can be seen as a filter or selective screen between the brain and reality.

> Closely related to the high-low-context continuum is the degree to which one is aware of the selective screen that one places between himself and the outside world. As one moves from the low to the high side of the scale, awareness of the selective process increases. Therefore, what one pays attention to, context, and information overload are all functionally related.
>
> (Hall 1976, p. 85)

3.5 Language

In everyday living, for a social animal like us, the way that we think becomes intimately tied up with the means we use to communicate - human cultural form, and particularly language. But it is a mistake to see culture and language as 'things', they are relationships, hence the critical importance of context.

> It makes no sense to talk about words or sentences unless the words and sentences mean something. For sentences to mean something, their components must be linked together in an orderly way. A linguistic expression must be encoded in some medium - such as speech or writing - for us to know that it is there. And there must be people involved in all this to produce and receive linguistic messages.
>
> (Baron 1986, p. 10)

Human language is therefore the web of relations constituted by meaning, syntax (linkage), medium, expression, and participants. Other factors, such as purpose and experience, come into the mix, but these can be seen as sub-factors. For example, the purpose of a message is a part of its meaning, and experience is usually what is being expressed by or represented in the message.

Given the communicative aspect of language it is not surprising the ‘language’ became the metaphor for the means used to ‘communicate’ our designs to machines. However the communicative aspect of programming is probably not the basis of the difficulty that many people have in learning to program.

> The hardest part of programming languages isn’t learning the programming languages themselves. It is figuring out how to recast your problem into programming language syntax. (Remember how the hardest part of algebra was putting “word problems” into equations, not solving the equations themselves?) ... There are several ways of doing this formulation. The most tempting (but usually the most disastrous) is to stare at the problem and then start writing lines of computer code. This strategy is like deciding to film a movie without first having a complete script. A few directors such as Federico Fellini can pull off this feat - and so can some experienced programmers. For most of us, though, the cost overruns of plan-as-you-go are enormous.
> (Baron 1986, p. 24)

The process of “figuring out how to recast your problem” is classic creativity, problem solving, not just the using of language for communication. In a sense at least, communication with others, or even the computer at this stage, is irrelevant to the actual solving of a problem, the only factor that is relevant here is meaning - you must know what the problem specification *means* before you can even begin to see the possible solution, let alone communicate it. Programming is mostly *creating* a solution to a problem not *communicating* it to the computer. So although ‘language’ is still the correct metaphor, seeing it as a thing, a system of communication, obscures the fact that, programming is mostly a process, creation. The purely linguistic nature of the solution, its *formal* representation, is irrelevant to the process of finding it except insofar as it assists rather than hinders the process. Linguistically, as an “interpreted formal system” (Haugeland 1985, p. 106), a computer does nothing but literally implement the syntax, that is, automatically make the formal moves. This makes a computer functionally equivalent to mathematics because it fits the following prescription; “If the formal (syntactical) rules specify the relevant texts and if the (semantic) interpretation must make sense of all those texts, then simply playing by the rules is itself a surefire way to make sense” (Haugeland 1985, p. 106). This is why obeying the rules of arithmetic guarantees a correct answer - ‘one makes sense’ in the limited means of the language, not in the wider sense of ‘understanding’ an aspect of the world. So the rules of the language themselves offer no assistance to the creative act of finding the solution.

Creation is different from calculation or computation in that the latter is fully deterministic while the former is not. The steps in creation are made by the human author *applying* the rules arbitrarily, so the order in the solution has two sources, the rules and the author’s choices (Haugeland 1985, p. 107). This is a way of saying that the language system, particularly the syntax, does not contribute to the creative process, it just represents the product of the creative process in a, more or less, formal notation. Natural language, even in communicative terms, is a largely informal system because it is a relationship between several components, not just a set of rules, a syntax. Moreover the participant component is a largely non-deterministic system, that is, undefinable by syntax. The result is that natural language is fundamentally a product of the need for humans to create *meaning*; the syntactic element in it is a byproduct of the communicative need for a congruence of meaning between participants.

Syntax, then, in natural language, supports the communicative not the creative process. Moreover, any language that is described entirely by syntax, as programming languages are, reverses the order of significance demonstrated by the elements of natural language - meaning before communication (syntax). Syntax is an emergent characteristic of the communication of meaning (Schoenemann 1999), so a 'language' that is entirely defined by syntax automatically disbars the most creative uses of natural language, metaphor, 'coining a phrase' and so on, and thereby causes "unnecessary confusion in the way we think about programs" (Backus 1978, p. 614). This is because normal thinking is *always* creative in some sense - either it is entirely new in content, or it is a new form (version) of content that you've considered previously, it's difficult to see how thinking could recur exactly, it is, by nature, ephemeral. If the 'meaning' of a statement is determined entirely by syntax (operational semantics) then it is entirely derivative, it can never be *original*[7]. Mental 'meaning', on the other hand is *always* original (Haugeland 1985, p. 87), the syntactic structure is added only as a means of making it communicable.

If the human condition can be captured in one idea then it is a search for meaning. Mind, as the web of meaning in the brain, is the source of human creativity, so it is the 'language of the mind' that we need to be concerned about in the consideration of the learning of programming, not the 'language of the machine'. Hence the creative aspect of programming forces us to consider many of the classic philosophical problems. Meaning, in the mental sense, arises from consciousness, I can hardly have a meaningful mental relationship with an aspect of the real world without being conscious of it. So the idea that consciousness is just an accidental side effect of computation, 'an epiphenomenon', denies the existence of 'meaning', and says, in effect, that the means by which the universe is made known is itself essentially purposeless (Penrose 1989, p. 580).

What we have is a biological-psychological continuum similar to Einstein's space-time continuum with the direct mapping of physical morphology to DNA at one end and the less direct, but no less powerful interaction between mind and reality at the other. The whole complex arises out of the drive for meaning. In a sense, life is just the expression of meaning at several levels. Being alive implies

[7]One would certainly be disconcerted, to say the least, if a programming language statement acted "creatively", that is, did not *always* cause the same basic operation to be performed. Maybe "Intercal" is the exception to this.

biological individuality, a radical separation of organism from environment, the 'intention' to be 'separate' from everything else. So what is meant by 'intention' here does not imply any conscious intent, just the 'intention' of process, of biological being. It is that self-defining process involved in being a system that is 'whole' in its own right. So, although a rock can be said to be 'whole' in some sense it is not the 'wholeness' implied by 'intent'. It does not encompass a process that works towards the goal of keeping it going, that guarantees its separateness, its 'wholeness'. A rock's wholeness is the simple totality of aggregation, there is no coherence provided by an internal self-defining system, no sense in which the rock as a whole is more than the sum of its parts. The meaning of life derives from this wholeness of being, of system coherence. It is more than any sense of meaning that can be imparted by aggregation, as is demonstrated by the fact that an organism can lose some of its 'aggregated wholeness', a limb for example, and still function as a whole system, albeit with some modification of its capabilities.

Wholeness is a property of systematic organisation. This is an obvious truism in the case of lving systems, but it also true of other systems in a more subtle sense and it is in this sense that Alexander defines 'organic order' as the role that the parts play in making something whole. "We define organic order as the kind of order that is achieved when there is a perfect balance between the needs of the parts, and the needs of the whole" (Alexander et al, 1975, p. 14). A system, any system, can be said to be a system only because it demonstrates this property of wholeness - it consists not only of all the parts that constitute it structurally, but some sense of being organised that is more than just its static 'structural organisation'. This functional organisation is a sort of 'purpose' or 'intention' that makes it a unit unto itself, apart from everything else. Such 'organisation for function' consists of the relationships between the parts. This is an important point because this is where 'meaning' comes into the picture - these relationships are the 'meaning' that each part has in terms of the whole system. Meaning is not inherent in any entity, it exists only in the relationships that make it part of a system seen as a functioning unit, so, for example, the inherently meaningless aggregation that is the rock discussed above, gathers meaning in terms of a larger system when it is utilised as a tool or a weapon. However that is not the end of the story because any system exists within a still larger system and this multiple layering is the source of complexity - a radical intertwining of multiple meanings. This is why classification is so fundamental to understanding. It's only when we have separated the layers, illuminated the hierarchy, that we can begin to understand, to see the forest for the trees.

This is how meaning is created in human terms, through relationships. Meaning is *not* a simple product of the thinking process, it arises from living, not thinking. The primary fact about being human is that it is a living system, an interaction between the human *object* and the environment. It is not possible to see a biological being in any sense other than this, as an ecological relationship - individual $\leftrightarrow$ environment, where the '$\leftrightarrow$' symbol indicates two-way interaction.

Everything about a biological being, its 'beinghood' in effect, derives directly from this relationship. Its morphological form is an expression of its evolutionary history encoded in its DNA, and its everyday behaviour is an expression of its individual history encoded in its nervous system. But on their own, DNA and nervous systems are formal means. If that's as far as it went then behaviour would be a simple case of innate response (instinct) modified by experience (learning). In other words creativity would be restricted to some simple modification of the stimulus-response relationship. However survival is the ultimate arbiter and driver of behaviour. It does not follow that the correct response in survival terms is always that derived by formal means, survival is not logic, it is contingency, and this requires adaptation, that is, in behavioural terms, learning, and ultimately creativity.

> Both genetic change and the process called learning (including the somatic changes induced by habit and environment) are stochastic processes. In each case there is a stream of events that is random in certain aspects and in each case there is a nonrandom selection process which cause certain of the random components to "survive" longer than others. Without the random, there can be no new thing.
> (Bateson in (Heims) 1991, p. 254)

The physical environment can be seen as an aggregation of substance with no meaning, it carries no information as such. Whatever happens in cosmic terms, just happens, it is a simple playing out of the physical rules established by the existence of the universe. But once you have organic life you bring into being 'meaning', where 'meaning' is the 'informing' of life. It is difficult to see life in any terms other than this - the 'intention' to embody the separation from everything else that exists. DNA 'informs' the separation at the physical level, it supplies the 'information' needed to define 'being a cat', or 'being a tree' in distinction from everything else. Being 'something' implies that 'something' is not just a part of the whole physical aggregation of matter, and that whatever it is that defines its separate identity, spirit or life-force, is the meaning of that 'something', its 'intention' to be different, to be an 'individual'. Information can't exist before the 'intention' to define difference. If things just happen without any effect on an 'organism' then they cannot be said to *mean* anything. Meaning requires a 'receiver', it is a relationship between separate entities, things that are more than just parts of the total aggregation (the cosmos), that have a 'wholeness' (being) that is complete in its own right and that can relate to the environment in 'meaningful' ways.

I find the contemporary notion of information as existing in pure material existence a rather strange one, if the term is meant to imply an association with 'meaning' - "objects can be used to store and transmit data, but information is not meaning" (Freeman 1999, p. 201). It is always system that gives meaning, that supplies the 'information' contained in a situation. The only 'information' that can be associated with a rock, for example, is that supplied by its involvement in a larger system. In other words, information is a product of a system, it is provided by the relationships between components of a system, not those between parts of a totality. Patterns can be seen in these terms as recurrences in the functioning of a system. So the rock, if it has crystalline structure, can contain 'information' derived from the magmatic, metamorphic or sedimentary system that produced it because the crystalline structure implies more than the simple aggregation through coincidental association, it says something about the chemical composition of the material that became the rock.

Meaning at this level, if it can even be called that, is purely structural, it is, in effect, the 'fossilised' patterns of system dynamics. In terms of the system which produced it, this is syntax, a structural pattern, not semantics. Real meaning involves a functional relationship based on some process, it "has a focus at some point in the dynamic structure of an entire life" (Freeman 1999, p. 18) where life in this context, I would add, is a wider concept than "biological life", indeed more like the sense in which Alexander uses it in attempting to explain his *quality without a name* (see Alexander (1979), p. 30). The meaning of an object or event is largely a reflection of its impact on the functioning of the larger system, of utility. If the system has no purpose, no 'intention' then it's difficult to see that the idea of 'meaning', or even 'information' is really applicable. These ideas imply a commitment on the part of the system to the relationship that carries the information or meaning, the utility of the object in the system's 'intention'. In other words, a relationship that creates 'meaning' is useful in some sense.

> I have said that a tool is only one example of the merging of a thing in a whole (or a gestalt) in which it is assigned a subsidiary function and a meaning in respect to something that has our focal attention. I generalized this structural analysis to include the recognition of signs as indications of subsequent events and the process of establishing symbols for things which they shall signify. We may apply to these cases also what has just been said about a tool. Like the tool, the sign or the symbol can be conceived as such only in the eyes of a person who relies on them to achieve or to signify something. This reliance is a personal commitment which is involved in all acts of intelligence by which we integrate some things subsidiarily to the centre of our focal attention. Every act of personal assimilation by which we make a thing form an extension of ourselves through our subsidiary awareness of it, is a commitment of ourselves; a manner of disposing of ourselves.
> (Polanyi 1958, p. 61)

The real point of consciousness is that adds another layer of relationships to the layer expressed in biological evolution. Evolution is the interplay between DNA and the environment and it allows lines of development that are simply not possible in a purely physical system. Similarly moving from the biological to the mental layer allows the emergence of relationships like conscious awareness that are not possible at the former level. So just as the relationship between DNA and morphological form is more than a direct causal one - there would be no room for evolutionary adaptation of morphological form to the environment if form was the product *only* of the DNA pattern - so the mental relationship level must be based on more than *just* neuronal patterns, there has to be feedback mechanism to relate neural organisation to what's going on in the world, hence consciousness and mind are emergent from the interaction between brain and environment just as morphological structure is emergent from the interaction between DNA and environment.

3.6 On Creativity

There are some ideas, some effects, that can only be explained by creative means rather than by purely scientific deduction. One can think of no possible experiment, other than Einsteinian type gedankenexperiment, or thought experiments, to explain the implications of, for example, time dilation effects, or the rewinding of time to discover the singularity. Many discoveries are only possible via the use of imagination. Someone had to think of the 'Big Bang' before we could begin to understand its effect on the current state of the universe. Similarly, the precondition for the discovery of the New World in the 16th Century was the ability to imagine the possibility that the earth is round rather than flat.

If the essence of creativity is not a fundamental freedom to express oneself, then it is hard to imagine what it is. The operative word in terms of creativity is surely freedom. Furthermore, this insight is backed up by the obvious fact that creativity, in terms of play, decreases as children approach maturity. That is, children are essentially free from the strictures of convention and responsibility, and this freedom is expressed in creative play. The mark of creativity is freedom to imagine - think for a moment about those situations in which, as an average person, you feel most free to express yourself. Surely this is those occasions when one is casually mulling over ideas in one's head. Experience tells us that the most freely creative situation in our lives is mental play, those times when we are engaged in what I wish to call 'free form thinking'. I do not mean to imply that this type of thinking does not have a goal or purpose, that it is not directed at a definite end. Rather, the notion of 'formlessness' refers literally to a lack of 'form' in the sense of a formal syntactic structure. This lack of form comes about, not because an absence of rigour in terms of purpose or direction but because of freedom from the need to communicate. At least in the early stages of such a process, the goal is to 'create' meaning in one's own mind, not in the mind of others, that can be taken care of later - for the moment we are engaged in thinking about something that concerns us just for the sake of thinking about it and for no other reason.

The outstanding feature of such moments is the complete lack of attention that we give to the formal structure of language, the point being that this formal aspect derives from the need for convention in the interaction between two communicating entities, the requirement to make what we say, or write, intelligible to others. When we indulge in free form thinking - thinking just for the sake of it - the communicative requirements of language are the furthest thing from our mind. We just go for it. Ideas are free to float into our field of thought, the connections are those between the meaning of the ideas concerned, semantics, not the syntax of word combination, so it is the patterns of experience, pattern language, that sets up the 'creative field'. Because we are not concerned with making our thoughts communicable we are unconstrained by the conventional rules, the syntactic structure, so this is language at its most creative - literally a free association of ideas.

This points to the reason for the difficulty we have in expressing creativity in programming - "too much of the computer has been used, and designed, as an exclusive extension of the formalistic capabilities of humans" (Waisvisz 2004). Thinking in terms of the code constrains us because the code is about 'communicating' with the machine, this is a formal language in the strictest sense possible - it is thinking like an automaton. And as Dijkstra says "the tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities" (Dijkstra 1982, p. 129), suggesting that the coding environment is not one in which the free association of ideas is appropriate.

In fact such free-thinking in the coding situation is nothing other than that phenomenon generally known as 'hacking'. But the point about 'hacking' is that it is not a problem if it occurs in an environment of free flowing thought unconcerned with formal communicative requirements. Hacking, in this situation, is just pure creativity. Children, when they first start to learn a language, use words in exactly this way, as meanings, not formal symbols. Because they are unaware of the formal structure of language they effectively ignore it and just express whatever ideas pop into their head. Draft versions of a piece of writing, especially early ones, are more like this too, the polishing process is, partly at least, the application of the filter of the formal requirements of the language being used. So it is not even the impulse to communicate that is the problem, both the child and the writer are ultimately motivated by that, it is the formal nature of communicative media, syntactic structure, that is restrictive of the free flow of thought that underlies creativity. What we want in creative situations is semantic, not formal, syntactic, structure - we want meaning. No wonder hacking in code doesn't work! As Polya said, talking about solving problems generally, way back in 1945, "the worst may happen if the student embarks upon computations or constructions without having *understood* the problem" [emphasis in original] (Polya 1948, p. 5).

But if it is true that a programming language performs as a straitjacket for creativity then the fact that most programmers usually work directly at the coding level would seem to contradict this. However, such practice is normally indictive of experience. The experienced programmer, like the experienced driver, has 'internalised' the conceptual processing, so that it might look as though no abstract thinking is occurring and it's all just a flow of 'doing', writing the code, or driving, virtually unconsciously. Of course, this has to be an illusion, one way or another the program is being 'designed', either consciously or by default, just as the car is being driven whether or not the driver is consciously concentrating on the act of driving.

> Many contemporary methods view design as a phase intermediate to architecture and coding, instead of viewing architecture and coding as products of design. But from a more practical point of view, we can't separate design from either architecture or implementation. If design is the activity that gives structure to the solution, and if architecture is about structure, isn't "design" a good term for the activity that produces it? And much of the code is about structure as well. Why shouldn't that be "design" as well? If you look at how real programmers work, you'll find they really don't delineate architecture, design, and implementation in most application domains, regardless of whether the official house method says they should or not. (How many times have you completed the coding before holding the review for your design document?) Object-oriented designers gain insight into the allocation of responsibilities to classes by coding them up. Empirical research on the software design process reveals that most developers have at least partially coded solutions in hand at the time of their design review and thus design decisions continue into the last throes of coding.
>
> (Coplien 2000*b*, p. 35)

So if it is true that one way or another the program is being 'designed' then the problem is not that there is no design being done, but that the design that *is* being done by default produces poor software. This observation is backed up, moreover, by ample evidence of the continuing crisis in software development. Most people involved in software development are experienced programmers, yet many projects get into difficulty as creativity is essential at several levels simply because of the nature of large systems. Firstly, the need for automation usually arises in large and complex organisations. Expressing what is required is itself a creative act. Secondly, and most pertinent to the programmer, the requirement specification needs to be translated into a plan of the system that meets the requirements, another creative act. All this creative activity needs to happen before coding even begins, the point being that it is part of the development of the program considered as a whole. Not only is the experienced programmer working with a higher level of understanding of the programming language, but starts with a preconceived notion, a plan, of the project that, at least in part, informs the writing of the code. It's no wonder that, to the casual observer, it looks like coding without thinking, without conscious design.

In a sense, too, the fact of the crisis in software development confirms the disjoint between creativity and communication. Both the specification of the system and the plan for building it are communicative in spirit because the people who require the system are different from those who are to build it, and those who do the planning are, more often than not, not the same people who do the actual coding. Therefore the need to communicate precisely is paramount in these situations. In this case the trouble is that natural languages are not formal enough, leading to ambiguity and lack of clarity. As Dijkstra points out we simply don't yet "know how to manage the design (or should we say 'the discovery'?) of software" (Dijkstra 1982, p. 222), 'discovery' being the operative word here. Again it is the mismatch between creativity (discovery) and formal methodology (management of the design process) that lies at the root of the problem of developing software successfully. "In spite of its name, software engineering requires (cruelly) hard science for its support" (Dijkstra 1982, p. 130). Dijkstra here is calling for support for the notion of what he calls the "intellectual individual". "With their stress on the supposed virtues of group activity (and on the need for "communication skills"!) they [Computer Science faculties] seem to regard minimization - or possibly even elimination - of his role as an ideal worth to be pursued. I regard that as a threat to our civilization" (Dijkstra 1982, p. 220).

Mind is a product of social interaction, so communication is an important factor, but this is a two way street. As much as mind is a product of interaction, interaction is a product of mind. Djikstra's point is that you can't use one to eliminate the other as they are interdependent, a group is made up of communicating individuals, it's a collection of minds - 'intellectual individuals' in Dijkstra's terms. At least part of the problem is caused by the fact that engineering is probably the wrong metaphor for software development as it suggests the idea of support type activities such as the calculation of stress values and the like that are ancillary to design in other fields, not central. Such factors are ancillary because they deal with characteristics of the domain concerned, materials science for example, rather than being characteristics of design (Sargent 1994).

Furthermore, designing anything "is not so much a process of careful planning and execution as it is a conversation, in which the conversing partner - the designed object itself - can generate unexpected interruptions and contributions" (Winograd 1996, p. xxi). Engineering is closer to the process of using formal logic than it is to writing creatively - it's like using natural language to write a technical manual rather than a poem. The means are the same but the end qualifies and formalizes the means. The fundamental issue underlying all the problems facing software developers is conceptual not technical (Tekinerdogan & Aksit 1999, p. 1). Even patterns, applied in the spirit of engineering - that is, X tells me to do this (where X is a principle, a rule, a template, or even a pattern) - are dangerous. Using patterns does *not* imply that "you no longer have to think. It just helps you decide what to think *about* [emphasis in original] (Shalloway 2003, p. 15).

Design is the primary issue in software development - how do we assimilate human creativity with mechanical automatism? As Blum says, design is a "contingent process" requiring "perpetual discovery" (Blum 1996) and this is not addressed by the application of any sort of engineering principles, even the recipes of "knowledge engineering" (Hofstadter 1985, p. 637). Understanding is the key, not knowledge alone, and understanding comes more from doing than from knowing. The idea that understanding can be built by a process of 'engineering' knowledge seems a very strange one given that 'engineering' is largely the application of known principles in a given situation - engineers "make new things and make old things better" (Discover Engineering 2005), turn ideas into reality, if you like. Putting a human on the Moon, for example, was an engineering problem, not a problem of 'understanding'. Understanding implies the ability to create new ideas, to use knowledge to comprehend novel situations not just to apply it to well-defined cases. "The payoff for understanding comes not in direct application of the newly learned material, but rather in transfer to new situations" (Mayer 1981). Programming is more about 'understanding', creating meaning - one *designs* a solution more than one *engineers* software. Knowing the rules of chess does not make you master player, the engineering or superstructure of the game and the skill of playing it lie in separate domains.

Perhaps it is the case that a lot of the current problems with software development in general are due to the fact that software is "not designed at all, but merely engineered" (Kapor 1996). Software, strictly speaking, is the experience of using the artefact, its 'design', not just its 'engineered' aspect, its raw structure. As developers we seem to miss the holistic nature of the artefact in the rush to build it - it is something that is built to be used by humans not something that is just built. As Winograd puts it, we tend to overlook the fact that "the designer stands with one foot in the technology and one foot in the domain of human concerns" (Winograd 1996, p. xx). This means that the process of designing something is more negotiation than rule-application, more dialog than program. In the end "it could be that the only thing common to all design is the intention to produce something useful. That does not mean that design theory and methodology research ends - it means that it is unending" (Sargent 1994).

Chapter 4

The Source of the Difficulties

Those who have attributed the preeminence to logic, and have thought that it afforded the safest support to learning, have seen very correctly and properly that man's understanding, when left to itself, is deservedly to be suspected. Yet the remedy is even weaker than the disease; nay, it is not itself free from disease. For the common system of logic, although most properly applied to civil matters, and such arts as lie in discussion and opinion, is far from reaching the subtility of nature, and, by catching at that which it cannot grasp, has done more to confirm, and, as it were, fasten errors upon us, than to open the way to truth.

(Francis Bacon 1561-1626)

We face, then, two great stochastic systems that are partly in interaction and partly isolated from each other. One system is within the individual and is called learning, the other is immanent in heredity and in populations and is called evolution. One is a matter of the single lifetime, the other is a matter of multiple generations of many individuals. The task is to show how these two stochastic systems, working at different levels of logical typing, fit together into a single ongoing biosphere ...

(Bateson in (Heims) 1991, p. 254)

4.1 Art and Science

We presume, in the modern age, that we are caught on the horns of a dilemma peculiar to our time. But, in fact, Plato has Socrates state the dimensions of the problem in "Phaedrus".

> If one could obtain by art, the power or capacity of these two kinds of operations, which in this instance we have performed by mere chance,

> it would be not unpleasant. ... To collect together a multitude of scattered particulars, and, viewing them collectively, bring them all under one single idea, and thereby be enabled to define, and so make it clear what the thing is which is the subject of our inquiry. ... [And] to be able again to subdivide this idea into species, according to nature, and so as not to break any part of it in the cutting, like a bad cook. ... Being a lover of these compositions and decompositions, in order that I may be able to speak and to think; if I find any one whom I think capable of apprehending things as one and many, I run after him and follow his footsteps as I would those of a god. Those who can do this, whether I call them rightly or not God knows, but at present I call them dialecticians.
>
> (Plato as translated by John Stuart Mill (Mill) 1946, pp. 92-3)

One of the major blindspots of modern thought is the neglect of the unifying impulse expressed in the idea of philosophy. For what is “love of knowledge” but ‘understanding’? The original philosophers did not envision the chasms in the edifice of knowledge that were to come. But this is not even as simple as the split between Science and Art, or Science and Philosophy, it is reflected in the very way that we see the world, the split between ‘intention’ and ‘knowledge’, the ideal and the real, morality and pragmatism. The idea that understanding grows out of a discourse with the current state of understanding, Plato’s dialectic, implies creativity as well as analysis, ideas as well as reality, spirit as much as reason. It shows an appreciation of “the importance of ideas - of ‘culture’, to use a modern term - for the forming of our minds” (Popper 1977, p. 166), where ‘mind’ is taken to be the result of ‘purpose’. “Mind, or thought, or reason, he [Socrates] decided, always pursued an aim, or an end: it always pursued a purpose, doing what was best” (Popper 1977, pp. 169-70). As Socrates goes on to say in “Phaedo” about the excitement that was aroused in him by the announcement of a book which taught that the mind (nous) “orders and causes all things” (Popper 1977, p. 170), one’s path through life is as much a matter of choice as of structure.

> What expectations I had formed, and how grievously was I disappointed! As I proceeded, I found my philosopher altogether forsaking mind or any other principle of order, but having recourse to air, and ether, and water, and other eccentricities. I might compare him to a person who began by maintaining generally that mind is the cause of the actions of Socrates, but who, when he endeavoured to explain the causes of my several actions in detail, went on to show that I sit here because my body is made up of bones and muscles; and the bones, as he would say, are hard and have joints which divide them, and the muscles are elastic, and they cover the bones, which have also a covering or environment of flesh and skin which contains them; and as the bones are lifted at their joints by the contraction or relaxation of the muscles, I am able to bend my limbs, and this is why I am sitting here

> in a curved posture - that is what he would say, and he would have a similar explanation of my talking to you, which he would attribute to sound, and air, and hearing, and he would assign ten thousand other causes of the same sort, forgetting to mention the true cause, which is, that the Athenians have thought fit to condemn me, and accordingly I have thought it better and more right to remain here and undergo my sentence.
>
> (Plato 1999)

From this it is clear that the loss of Plato's unified notion of art as "the power or capacity of these two kinds of operations", analysis and synthesis, this disastrous separation of his "one and many", the parts from the whole, results in rigidities of both mind and culture. It lies at the heart of many of the movements that have bedevilled us, materialism, the various flavours of determinism, relativism, subjectivism, modernism, post-modernism, and so on. These are all merely points along the course of the journey, stages in the evolution of understanding, not final positions. If we insist on seeing the word as the concept, the image as the subject, the book as the argument, the brain as the mind, the code as the program, then we are prevented from moving forward. Concept, subject, mind, argument and program are all expressions of fluid, creative, thinking, of art. We need word, image, book, brain and code as markers, representations of creative process, but they are not creative in and of themselves; they are artefacts not art.

> He then who thinks that he can leave behind him an art in a book, and he who learns it out of a book, and thinks he has got something clear and solid, are extremely simple, and do not know the saying of Ammon, or they would not suppose that a written book could do anything more than remind one who knows already. Writing is something like painting: the creatures of the latter art look very like living beings; but, if you ask them a question, they preserve a solemn silence. Written discourses do the same: you would fancy, by what they say, that they had some sense in them; but, if you wish to learn, and therefore interrogate them, they have only their first answer to return to all questions. ... There is another sort of discourse, which is far better and more potent than this. ... That which is written scientifically in the learner's mind. This is capable of defending itself; and it can speak itself, or be silent, as it sees fit ... the real and living discourse of the person who understands the subject; of which discourse the written one may be called the picture.
>
> (Plato translated by John Stuart Mill (Mill 1946))

Nowhere is the tendency to mistake the product for the process more pronounced than in science. Symbols necessarily have a characteristic that is not manifested in the entities for which they stand, an essential abstraction from the objects they represent. They thereby become amenable to mental operations that are independent of the physical existence of their subjects. This is the first step in the process that leads both to idealism and formal methodology. Mathematics and logic are games with symbols played according to fixed rules, but where, and this is the critical point, the symbols are not even really symbolic in the sense of standing for something (Weyl 1959, p. 55). The trouble is that since Gödel we are aware that in any non-trivial formal system there are elementary statements that are evidently true but which cannot be deduced within the system - “the fields of propositions accessible to insight on the one hand and to deduction on the other overlap, neither of the two being contained in the other” (Weyl 1959, p. 219).

If the logico-deductive basis of the scientific method is put in doubt in this way, so that there is no Hilbertian assurance of internal consistency[1], then we are left to justify science by means other than the power of deduction.

> The ultimate foundations and the ultimate meaning of mathematics remain an open problem; we do not know in what direction it will find its solution, nor even whether a final objective answer can be expected at all. “Mathematizing” may well be a creative activity of man, like music, the products of which not only in form but also in substance are conditioned by the decisions of history and therefore defy complete objective rationalization.
>
> (Weyl 1959, p. 3)

Ultimately it seems that the grand attempt to distinguish science from art by means of formal methodology is doomed, a victim of the “epistemic cut”, the split between the observer and what is observed. The point of Kuhn’s view is that “science [is] characterized more by the paradigms employed by scientists than by their methods of enquiry” (Novak 1977, p. 43). It is part of the pantheon of human creative endeavour, an exploratory, explanatory art, a mythology, in the original sense of a defining narrative, in other words.

> Man tries to make for himself in the fashion that suits him best a simplified and intelligible picture of the world: he then tries to some extent to substitute this cosmos of his for the world of experience, and thus to overcome it. This is what the painter, the poet, the speculative philosopher, and the natural scientist do, each in his own fashion. Each makes the cosmos and its construction the pivot of his emotional life, in order to find in this way the peace and security that

[1] “Hilbert’s dreams of reformulating classical mathematics as a formal axiomatic system equipped with absolute proofs of consistency and completeness were dealt a cruel blow by Gödel’s findings in 1931.

In his famous paper, Gödel proved that it was impossible to find a metamathematical proof of such a system’s consistency without employing rules of inference inexpressible within the formal system under consideration. (More precisely, Gödel proved his results of any axiomatic system comprehensive enough to contain the whole of arithmetic.)” (jt 2000)

> he cannot find within the all-too-narrow realm of swirling personal experience.
>
> (Einstein quoted in (Hoffmann) 1972, pp. 221-2)

The keys to scientific exploration, then, are simplicity, intelligibility and emotional satisfaction - an attempt to make experience accessible to explanation. Einstein put his finger on the real epistemological situation when he said:

> The historical development has shown that among the imaginable theoretical constructions there is invariably one that proves to be unquestionably superior to all others. Nobody who really goes into the matter will deny that the world of perceptions determines the theoretical system in a virtually unambiguous manner, although no logical way leads to the principles of the theory.
>
> (Einstein quoted in (Weyl) 1959, p. 153)

Scientific method is thus shown to be an aspect of the standard human drive to create the mental world that we know as the mind. It hardly needs any justification beyond that it contributes to understanding in the way that other human activities do. And if this means that it is once again seen as a branch of philosophy then this does not make it any the less significant. Indeed it might lead to a greater awareness of the need for synthesis, for a view of the whole sweep of that which is human, amongst those who consider themselves scientists. “Perhaps the philosophically most relevant feature of modern science is the emergence of abstract symbolic structures as the hard core of objectivity behind - as Eddington puts it - the colorful tale of the subjective storyteller mind” (Weyl 1959, p. 237).

> Traditionally there are two ways in which science can be justified, the Platonic and the pragmatic one. In the Platonic way - “l’art pour l’art” - science justifies itself by its beauty and internal consistency, in the pragmatic way science is justified by the usefulness of its products. My overall impression is that along this scale - which is not entirely independent of the Buxton Index - Europe, for better or for worse, is more Platonic, whereas the USA, and Canada to a lesser extent, are more pragmatic. ... But the answer is quite simple: in computing science the conflict need not exist - and that is what makes the subject so fascinating! - To quote C.A.R. Hoare - from memory - : “In no engineering discipline does the successful pursuit of academic ideals pay more material dividends than in software engineering.” I could not agree more.
>
> (Dijkstra 1982, p. 271)

In the final analysis science is no different from art, both enterprises are about making sense of experience, and both therefore encounter the main paradox addressed by this dissertation, the problem of meaning, the “epistemic cut”. It is true that “meaning emerges from context” and that “without a context, nothing makes sense" (Ferguson 1980, p. 303), yet we can only deal with the input of the senses subjectively.

> The basic claim of science is objectivity: it attempts, through the application of a well defined methodology, to make statements about the universe. At the very root of this claim, however, lies its weakness: the *a priori* assumption that objective knowledge constitutes a description of that which is known. Such assumption begs the questions '*What is it to know?*' and '*How do we know?*'.
>
> (Maturana 1970, p. 5)

4.2 Programming as an Art

If it is true that scientific objectivity is largely a façade, a mask obscuring the 'subjective storyteller mind', then it should not be surprising that it is difficult to teach as this is true of every creative medium. Even professors of literature fear to tread in the teaching of creative writing classes. In more scientific fields, objective facts and logical systems are, on their own terms, relatively easy to get to grips with at a basic level. Most of us, for example, can fairly easily learn the basic concepts of physics without ever becoming competent in the art of generating novel insight from them - we know enough to be able to read a story in physics, but we do not have the conceptual structure to 'free' our imagination enough to create a new story.

> Even those students with top marks in applied areas like physics and biology find they have the proclivity to pass tests, but experience difficulty putting their classroom knowledge to use in solving any practical problems in the real world.
> "What stands in the way is what I call the 'fact fetish'," explains James Paul Gee, author of What Video Games Have To Teach Us About Learning and Literacy. "For me, learning an area like biology should be about learning how to 'play the game' of biology, that is, learning to think, act, and value like a biologist."
>
> (Krotoski 2005)

A computer system is a much less complex entity, and the facts about it far less numerous, after all, than is the case with physics or biology, yet it still requires a deeper level of understanding to program it than one would initially suspect. Programming involves more than *just* the computer system itself, it is not entirely encompassed by the idea of 'computer science'. Engaging the 'storyteller mind' requires tapping into the whole of the psyche, to 'conceive' as well as to 'perceive'. Yet the language that we use to teach programming is totally unforgiving, it demands perfection.

> The computer resembles the magic of legend. ... If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.
> (Brooks 1983)

The programmer is a conscious agent, an organism with a mind, yet at its most basic level the language we give her to program with is the language of an unconscious automaton. No wonder it is a struggle to become proficient as a programmer. This is the language of determinism, not creative originality. There is no scope in the language itself for choice, for Kant's 'causality of freedom' or his 'faculty of determination'. This is not simply an engine of cognitive function, human mental performance is simply not constrained to formal means, it depends at least as much on emotional factors as on cognitive capacity. This is most noticeable in children - ask a child to demonstrate, for example, her skill at counting, and, if the whim takes her, she will deny the ability even though she has previously demonstrated it. We don't yet know how the emotional component of the whole personality works in terms of motivation of cognitive function but it clearly does affect performance. A language that addresses the computational performance of a machine is not likely to be effective in terms of the functioning of the emotional system, it is bound, indeed, to be entirely dumb in emotional expressiveness. This is a significant factor in considering the activity of programming because it seems highly likely that the ability to make choices and emotional state are closely connected. As creativity is mostly 'making choices', it is obviously conditional on, or at the very least, related to, emotional factors. Choice, in composing a program, is always made by the agent using the language, not the mechanism that implements it.[2] Because the language is implemented mechanically it cannot express freedom in the way that free flowing thought does.

Of course, freedom is a two edged sword - it is just as likely to lead to frustration as it is to satisfaction, and this would seem to be the problem for novices in particular. So although the programming language provides the means for making choices, it cannot, by its very nature - it implements a deterministic mechanism - provide assistance as to choosing the option that addresses a particular situation. In other words, it is an inherently uncreative form of language because the 'meaning' it gives expression to is the functioning of a machine - in the computer system, syntax is structure and semantics is function. This mechanistic aspect makes it the exact opposite of normal thinking because in human terms meaning is connection not direction. The task that a program is addressing, moreover, exists in the real world of human and organisational issues, not just the electronic system. So even if it addresses the virtual world of the computer perfectly it can fail miserably to meet the requirements of the situation it was contracted

[2]In fact, natural language is itself, to some extent, a mechanism - one that implements 'meaning' (more will be said about this later, see Section 4.5).

for. Using a language based *only* on the electronic pathways is like addressing a multi-dimensional physical system using Euclidean geometry, some connections are never going to be made. The language of formal logic, like the language of Euclidean space, expresses only a part of the whole system, human experience in the case of the former, and spatial relations in the latter. Designing a program is more about understanding what needs to be done in terms of the commissioning system as a whole, not just the mechanical component of it.

But this failure to express the totality of the situation is only a side effect of using a language based on mechanical means. Meaning is fundamentally emotional rather than simply functional in human terms, therefore using the word to denote what a machine does is highly misleading. Consider where meaning comes from - it comes from the reaction of the nervous system of a living organism to an event in its immediate environment, that is, it is a subjective response. The feeling of fear is what danger 'means' to an organism. But its subjective nature means that it is, strictly speaking, uncommunicable. I can no more 'feel' your fear than I can fly to the moon. You can tell me about, for example, your fear of heights, try to explain it to me, but I can never apprehend it directly. Meaning, at the objective level, is based on reason, it is definitional and explanatory. Yet even here it would be a mistake to discount its roots in the emotional system. Take the example of a person having an interest in a particular field, who finds, for example, pottery shards endlessly interesting. How else can one explain why a person is interested in one domain of information rather than another, other than by positing a personal, subjective and fundamentally emotional reason? - it is all ostensibly the same, just information. The big evolutionary step taken in the human case is the transfer of the idea of 'meaning' from the emotional to the intellectual system, to make it, eventually, the basis of reasoning not just automatic response.

> In the evolution of communicative behaviour, it is a common observation that the "raw material" of the behavior is a fragment of a motor pattern or an autonomic reflex which developed under selection pressures to serve needs that might be quite unlike communication. That is, these progenitors exist (or once existed) to serve more fundamental needs of the organism and were only secondarily adapted in the course of the evolutionary process to serve this communicative function. ... In that spirit, the original function for the externalizations or corporealizations of the psyche that we recognize as art (or artefacts of the creative process within) was to serve the individual, commonly to explore one's own thoughts and capabilities, to define one's self!
>
> (Greenberg 2004*b*)

The ability to reason is the means by which organic life goes beyond the possibilities available to an automatic nervous system, beyond the limits of 'instinct'[3].

[3] "Our genetic blueprint enables our brains and societies to live creatively in an uncertain world . . . As the ability to use language is ingrained in us, so to is our sensitivity to novel

It marks the border between the animal and the human mind. Without it most of the potentials that we think of as human would not be realised, even those that are ostensibly unreasoning. We don't normally think of love, for example, as being reason-based, and as the product itself of a direct line of reasoning a particular 'love relationship' probably isn't. Yet it's hard to envision how love could exist in a purely instinctive system because the capacity to put another's life before your own is entirely contrary to the instinctive response to danger[4]. Insofar as love can be said to override instinct it *must* be indicative of some higher order of brain activity than pure instinct.

It's difficult to believe that the two systems, instinct and reason, arose independently in the same biological system, the nervous system. Clearly one emerges from the other, or, more probably, they both emerge from a common ancestor, and this is suggested by their respective points of occurrence in the evolutionary tree. But the point here is that the connection between the two is the idea, and the power, of 'meaning'. At the most basic level of animal life there probably is no phenomena that can be labelled as 'meaning' in the abstract. The organism simply responds on cue. But this makes for a highly inflexible system, which is great if the environment in which the organism lives never changes, but deadly in an ever changing world. So the next step is to impose a flexible medium between stimulus and response, a means of making decisions. At first this is purely emotional, that is the decision-making is based purely on perception, the animal reacts 'emotionally' to the perception, but anything other than an automatic reaction is not possible without it. Once there is a layer of neural process rather than a simple link between the stimulus and response the potential to modify that process arises, and this leads, eventually, to human reason.

Reason is more than computation because it uses inductive process as well as deduction. The deductive process is automatic and mechanical - program-driven - whereas the inductive process is responsive to super-neural data (thoughts). Because reality never appears in perception as a coherent whole, the ability to construct a representation of it from both the current input from perception and memory of previous perception, to 'conceive' or 'create', is critical to reason. If knowledge was confined to what was currently available to the senses it would not be possible to 'reason' out a response, you would just have to do, more or less, what you've always done given similar perceptual input. This is often the right thing to do, in terms of survival, but not always, hence the evolutionary advantage. The ability to generate data (conceive), as well as to receive it as input (perceive) is vital for handling complexity without overloading the neuronal

conditions, without specifying the actual response [we should make]" (Richard Potts quoted in (Bower) 1997)

[4]Attempts to bypass this apparent impasse have been postulated on the basis of an instinctive 'loyalty' to the continuation of one's genetic heritage, rather than one's own personal life, but if the apparent indifference to personal danger is seen to include the 'protection' of individuals other than close relatives, then the argument reduces to 'loyalty' to species, or race maybe, and I doubt that most people would take altruism that far if it were simply based on "selfish genes".

system. The mind is the means by which the brain can make arbitrary, data-driven, connections rather than programmed, instinctive, ones. It can *create* stories to act out (simulations) as well as live out a prescribed script. As such it is an empirical discoverer through "subjective simulation"(Monod 1974, p. 145) as much as it is deductive. Real novelty emerges from the combination of concepts into new form not deduction which simply uncovers pre-existing information.

> In most lives insight has been accidental. We wait for it as primitive man awaited lightning for a fire. But making mental connections is our most crucial learning tool, the essence of human intelligence: to forge links; to go beyond the given; to see patterns, relationships, context.
>
> (Ferguson 1980, p. 32)

"Patterns, relationships, context" are the essential elements in the generation of new form, the ability to "go beyond the given".

Going beyond the given is the essence of solving a problem. The 'given' in any programming situation is the initial representation of what the program is meant to do, that is, the problem to be solved by the program. So the first, and most important task of the programmer is to literally create the conceptual 'problem space' in which the solution will be generated.

> First, the programmer must have a clear analysis of just what constitutes the task and the potential set of actions for the program. The task cannot be described with some vague generality like "diagnose illnesses" or "understand newspaper stories" but must be precisely stated in terms of the relevant objects of the environment and the particular properties that are to be considered. As we will discuss at length later, this task is the most critical. It results in the generation of a systematic domain, which embodies the programmer's interpretation of the situation in which the program will function.
>
> (Winograd 1983, p. 96)

This setting up of the 'problem space' in all its conceptual richness, is the task that Alexander characterises as "the task of making up the language from which you can later generate the one particular design" (Alexander 1979, p. 324), the point being that it is the programmer's understanding of the conceptual space in which the problem is situated that allows a solution to be generated. The understanding that is being sort here, the meaning being generated, is *not*, in any sense, that which is expressed by the execution of the program, the operational semantics, but the semantics of the larger situation in which the execution occurs. It is this fundamental fact that makes the programming language the *wrong* language in which to think about the generation of the solution because *the program* is merely a particular implementation of the solution. In any sensible reading, the solution is conceptual in nature. After all, the same solution can be implemented in many programming languages, so any particular program cannot be *the* solution, just an implementation, a particular representation, of it.

4.3 Learning to Program

A programming language statement has an operational semantics formally defined by the compiler. So any sequence of such statements, and therefore, any program, has only one possible outcome given the same input of data or user interaction. The difficulty with this is that it means that, given any level of complexity, the only ways to discover what a program 'means' is to compile and execute it, or to carefully hand simulate it statement by tedious statement. But this is like having to look up the dictionary definition of every word you encounter during a conversation, and presumes exactly one 'standard' definition for every word. Furthermore, despite the fallacious impression imparted by strict grammarians that there is a well-defined set of *parts of speech* and that every English word can be straightforwardly assigned to one or more of them, natural languages differ from programming languages in precisely this property. The grouping of words into the grammatical categories of natural language is highly dependent on the purposes of classification, and the word 'laughing', for example, can operate, at different times, as a verb, a noun or an adjective (Winograd 1983, pp. 49-50). Strictly speaking, syntax is about the act of communicating, not meaning as such - the structuring is important in *communicating* meaning. The essential relationship is syntax-facilitates-communication rather than syntax-defines-meaning.

Moreover the syntax-gives-semantics formula fits only the passive act of understanding, not active creation. It can only help you understand an already written program, so it is useless in terms of writing one from scratch. Creating meaning on the fly, so to speak, is infinitely harder than apprehending that which is already created, even given a strict correspondence between syntax and meaning. Operational semantics is just not a good fit with the way that the human mind works. Attribute grammars and the like may be useful in understanding operational semantics but they are hardly less obscure than the the compiler itself. Furthermore, they are themselves, formal systems, and therefore fit human-style thinking, particularly its creative aspect, just as badly as the programming language does itself. "Unfortunately, all the formal approaches to semantic definition [of programming languages] require a great deal of sophisticated effort and produce a result which is impossible to read without extensive study" (Horowitz 1983, p. 36).

Creativity is a necessary response to the existence of infinite possibility in experience. For the human, unlike the robot on a production line, the corporeal nature, and the spatial and temporal location of the objects it has to deal with are not precisely delimited - the human has to be able to locate and handle all types of objects and events wherever and whenever they appear. "The brain doesn't control the trajectory of the joints; rather, it initiates that trajectory and controls the material properties of the muscles" (Rolf Pfeifer quoted in (Spinney) 2005 p. 18). Clearly if the brain had to 'compute' all the precise values of distances, angles, weights, textures and solidity involved in locating and handling objects, nothing would ever get done (Pinker 1997, p. 11). As humans we 'create' the response to a given situation by means of previous experience, that is, our particular behaviour is 'patterned' on what has worked for us in the past.

It is in this sense that one can claim that a pattern language underlies all of our activities, we are organisms dealing with the essentially unpredictable guided by experience, not precision mechanisms in closely defined, and therefore predictable, situations, propelled by precise computation. Even in the case of the robot, the need for exact computation of *every* single value has to be avoided. We *pattern* the input as much as possible - that's what production-lining a process does - so that the computations can be done in advance of the activity. In some sense, then, even robots use a pattern language of sorts. But, strictly speaking, a machine has to be presented with a program, a formally defined operational semantics, precisely because it doesn't 'understand', it creates no 'meaning' of its own. 'Understanding' can be regarded as *a means of avoiding direct 'real time' processing of input data*, and it does this by basing all activity on a pattern language. The mark of an organism is a pattern language, that of a machine, a formal one.

There is a difference between causation and strict determinism and it lies in the fact that no *real* system is entirely closed, the point about closed systems is that they are formal and *abstract*. So all real systems interact in some way with all other systems, the universe would not be universal otherwise. But this limits the effectiveness of analysis as even the strictest analysis of a system cannot predict some unexpected interaction with another system. As carefully as I plan out my day's activities, I cannot say that my actions are determined in advance. Strictly speaking, causation sets up the conditions for an event, rather than directly causing it. Causation is context-setting rather than event-directing - the mass of the sun sets up the conditions for the Solar System, warps space-time locally, rather than determining any particular configuration of planets. Strictly speaking, there are no forces causing action, but fields setting up the *potential* for action to occur.

What instinct means is that the connection between perception and behaviour is 'hardwired' into the brain, it is strictly determined, hardly adaptable at all. If the mark of intelligence is adaptability, then this is both an advantage and a disadvantage in terms of survival. Often, instinct is the right way to go. After all, evolution has invested a lot of hard won experience into the genome, and this might help to explain effects such as the possible prescience of earthquake activity that many animals seem to exhibit[5].

[5]This is mostly conjectural because it is difficult to assign a particular 'cause' to any animal behaviour because the animal can't tell you what it is that is 'disturbing' it. Nevertheless

Of course, generally speaking, the big advantage of individual adaptability, as opposed to adaptation at the species level, lies in responding more flexibly, and therefore usually more effectively, to experiences that lie outside of the species' 'normal' range of experience where biological selection has not had time to adjust the hardware, so to speak. This is the explanation for those features that Hofstadter identifies as "defects" of the mind (Hofstadter 1979, p. 302). Individual adaptability can only work within the confines of the material that biological adaptation has provided, the mind can only use those structures and capabilities that the brain has evolved. And if this greater flexibility has made intelligence 'noisier' in terms of direct survival-specific information, this might explain why some basic instinctive drives for food, security, sex and so on, are so insistent in humans, to the point of pathology in some cases - they have to be in order to get their fair slice of the easily diverted attention of an organism that has highly 'specialised' (or should that be 'generalised'?) in creative curiosity.

Living is a process, *a way of being* in the world. At heart it is the creative interplay between an organism and environment. So your mind is literally your way of being in the world. Most humans in the same, or similar, cultural setting will approach a pretty standard model, that is, they will see the world and act within it in an extremely similar, although individually unique way - they will follow the same patterns. Notice that patterns are not in the same vein as 'rules', 'pathways', or even 'principles', as is shown by the fact that one can be individually unique within the common substrate of patterns. This is because life, being stochastic and therefore informal, is a much less rigid process than any mechanical, and therefore formal, one. The fact that one's patterns of living are tied inexorably to mind can be observed in those cases where a person's way of being in the world is *not* close to the 'standard model'. Because all the means of measuring intellectual ability derive from the cultural norm of western society, patterns of behaviour that are, to the people concerned and those close to them, "completely natural and creative and, in fact, quite inspired ways of perceiving the world, [are] rendered bizarre and meaningless" (Klotz 2004). But notice that this mismatch is caused by the lack of meaning imposed by seeing the 'normal' as rule-based (formal) rather than pattern-based (informal).

So the heart of the difficulty in learning to program is the difference between the way that a computer works and the way that the human mind works, and one can't even approach this difference without reference to various issues raised in philosophical and psychological discourse. The very word, 'psychology', expresses the split between mind and machine, though probably unwittingly. 'Psyche' means spirit, freedom expressed through intent and purpose, and 'logos' is logic, the 'immanent', a strict flow of cause and effect, often seen as divine. But the spirit, the life-force, is not logical. It emerges from the so-called 'logic of survival'[6]

Chinese authorities claim that observations of unusual behaviour in animals contributed to the decision to order an evacuation of the city of Haicheng, population approximately one million, just prior to a 7.3 magnitude earthquake in 1975 (Mott 2003)

[6]Strictly speaking survival is clearly *not* a matter of logic. It is not possible to 'logically

through biological evolution, but it gives rise to a new stream of development, the ‘evolution’, loosely speaking, of the individual personality - psychological or spiritual evolution - which can, and often does, run counter to the strict ‘logic of survival’.

Programming is therefore the construction of a logical system using a creative system, the mind - quite a disjoint when considered carefully. The mind works by weaving together threads from many diverse strands, rather than following a single logical strand, so forcing it to be strictly logical is not its natural bent. Pattern is the basis of mind, not logic. We are essentially *social animals* not Minsky’s “meat machines”. The crux of the matter, for an animal, is biological survival, and that for a social entity is social interaction. Both of these involve participation in informal milieu rather than formal systems, so the mind that has evolved in this evolutionary matrix needs to deal with probabilities rather than known certainties. The “patterns of feeling” (Langer 1962, p. 98) that we use to deal with uncertainty and complexity are simply not reducible to logic. The ancient Greeks were right: “there is no order in a purely chance event. To find order in chance - to discover a mathematical pattern - you have to see what happens when the same kind of chance event is repeated many times” (Delvin 2000). Dealing with uncertainty, then, is a matter of discovering the patterns of experience.

> People perceive patterns anywhere and everywhere, without knowing in advance where to look. People learn automatically in all aspects of life. These are just facets of common sense. Common sense is not an “area of expertise”, but a general - that is, domain-independent - capacity that has to do with fluidity in representation of concepts, an ability to sift what is important from what is not, an ability to find unanticipated analogical similarities between totally different concepts (“reminding”, as Schank calls it).
>
> (Hofstadter 1985, p. 640)

In the end, the brain is just the biological device that makes experience meaningful, that makes the experiential history of the organism available to its current state of being. As the sage said, “those who forget their history are bound to repeat their mistakes”, except, one might add, the fatal ones.

> There is no such thing as a computational person, whose mind is like computer software, able to work on any suitable computer or neural hardware - whose mind somehow derives meaning from taking meaningless symbols as input, manipulating them by rule, and

prove’ by formal means that a particular response will result in survival. The only way to ‘prove’ that a particular response will result in survival is to try it and succeed in surviving, but it is important to note that this is a not a general ‘proof’, it applies only to the particular circumstances in which it was tried. This is why pattern languages are more effective at this game, they inform response (that is they are ‘informal’) rather than dictate (formalise) it. They operate outside of Gödel’s limit to what can be proved using formal language.

> giving meaningless symbols as output. Real people have embodied minds whose conceptual systems arise from, are shaped by, and are given meaning through living human bodies. The neural structures of our brains produce conceptual systems and linguistic structures that cannot be adequately accounted for by formal systems that only manipulate symbols.
>
> (Lakoff & Johnson 1999, p. 6)

Of course, the idea that a computer is a formal system that manipulates symbols, leads also to the equally common fallacy that it can be considered an unbiased, impartial observer. Somehow, it is seen as a brain minus emotions, a 'brain' that deals in facts alone. A "computer, lacking emotions, hunches and prejudices, would consider and generate only the stark facts of the matter" (quoted in (Winograd) 1987, p. 156). It has even been said that computers should replace judges because, "unless programmed to be biased, [they] will have no bias" (quoted in (Winograd) 1987, p. 156). These ideas show how far the brain-computer confusion has spread. The fallacy lies in the notion of a computer as equivalent to a human observer - but freed from any personal view. Of course, a computer observes, considers, and decides nothing, it does not even 'generate the stark facts of the matter' - these are not functions of formal systems, they are functions of 'understanding' systems.

The reason that the brain is *not* just a computer is precisely because what a computer implements is formal means while a brain implements meaning. The driving force in one case is syntax, in the other it is the balance between freedom and necessity, choice and survival. So the language involved in each case is quite different. The language of a computer is formal logic, it implements a program, that of a brain is pattern language, it implements life, and life is about learning. If learning could be programmed then education would not be problematic, we could just implement the appropriate program - it's called 'rote learning'. But what we know from history is that it doesn't work. Even in the case of computer programming itself, experience tells us that you can't 'program' a human to program. Humans are thinking beings, not programmable machines, they run on patterns, not instructions. No computer ever solved a problem that cannot be solved by the application of fixed rules because such "a problem cannot be solved at the same level of consciousness as it was created" (Einstein). Yet the essence of being human is the growth of consciousness, the solving of problems that one has never encountered before, and that is why programming is so difficult.

A programming system is *always* a limited description of the *domain in which the problem to be solved exists*. It expresses only a limited number of the potentialities, the entities and relations, that exist within the problem domain. The surprising thing is that, given these limitations, programming has proved to be as powerful as it has. What this formula is really saying, of course, is that the use of the programming system by people has been amazingly creative, and has made it easy to forget that this is a limited system that is being used. Douglas Hofstadter makes the point in "Gödel, Escher, Bach" that these limitations might not apply to a computer system that allows multiple levels of knowledge that include a level of description that applies to itself. But he does admit that his contention is unsupported intuition (Hofstadter 1979, p. 152) and, in any case, this is not true of programming environments. The programming language constrains our thinking to only those mechanisms that it provides - a fixed level of consciousness in Einstein's terms. However understanding is not just a fixed relationship between the representation and the entity being represented, it is dynamic ever-changing one that permits new levels of consciousness to arise, and this is why 'understanding systems' (human beings) are capable of solving problems.

As biological beings we are forced to act in many situations regardless of our own individual propensity to do so. Heidegger calls this property 'throwness', we are 'thrown' into situations which force us to act, and it is this necessity to be ready for the unexpected that lies at the heart of our intelligence, not the fact that we can manipulate symbols. So phenomenon like reflection and abstraction are important in, but do not constitute the basis of, everyday action. Whenever we use abstraction, that is, build a representation of a situation using an analysis of the objects and relations within it, we also create a corresponding 'blindness' - our thinking is limited to what can be expressed in terms our representation. Reflective thought is impossible without abstraction but its limitations, the associated 'blindness', must be kept in mind.

> The essence of our intelligence is in our 'thrownness', not our reflection. Similarly, Maturana shows that biological cognitive systems do not operate by manipulating representations of an external world. It is the observer who describes an activity as representing something else. Human cognition includes the use of representations, but is *not based on representation.* [Emphasis added] When we accept (knowingly or unknowingly) the limitations imposed by a particular characterization of the world in terms of objects and properties, we do so only provisionally. There always remains the possibility of rejecting, restructuring, and transcending that particular blindness. This possibility is not under our control - the breakdown of a representation and jump to a new one happens independently of our will, as part of our coupling to the world we inhabit.
>
> (Winograd & Flores 1987, p. 99)

Logic, and therefore programming languages, limit our thinking to the fixed syntactic relationships of the representing system, in a sense we are forced *not* to think, at least not to think in the way in which that word is normally understood. But for true understanding we need the creative potential implicit in the idea of language as a dialog that permits new relations to emerge, for consciousness to grow. So programming is difficult *because* it limits our intelligence, makes us partially 'blind' so to speak, not because it is the use of a system that is, in itself, inherently difficult.

> The essence of intelligence is to act appropriately when there is no simple pre-definition of the problem or the space of states in which to search for a solution. Rational search within a problem space is not possible until the space itself has been created, and is useful only to the extent that the formal structure corresponds effectively to the situation. ... It has long been recognized that it is much easier to write a program to carry out abstruse formal operations than to capture the common sense of a dog. This is an obvious consequence of Heidegger's realization that it is precisely in our 'ordinary everydayness' that we are immersed in readiness-to-hand. A methodology by which formally defined tasks can be performed with carefully designed representations (making things present-at-hand) does not touch on the problem of blindness. We accuse people of lacking common sense precisely when some representation of the situation has blinded them to a space of potentially relevant actions.
>
> (Winograd & Flores 1987, p. 98)

4.4 Programming *equals* Designing a Program

So a programming system constrains our imagination by reducing us to using only those concepts that it provides. But you can't imagine anything that isn't based on concepts that you already have in your head and solving a problem requires 'imagining' the solution. So the fundamental difficulty caused by trying to learn how to program using a programming language is that you have to 'unlearn' how to think, to become a logic machine rather than a thinking being. A being has to survive, not just continue to exist, and this implies autonomous rather than automatic behaviour. You just can't just do what you've always done in a given situation in a *programmed way*, you have to do it in an *intelligent* way, and this is the difference between a program of set instructions and a language of tried options from which you *create* a strategy. In the end meaning derives from living, from experiencing life. A machine has no meaningful relationships with its environmental substrate because it has no being, no *élan vital* - it doesn't *care*. Intelligence derives from a being's attachment to life, so meaning is fundamentally an emotional relationship. The mind is built on patterns of meaning not on neuronal circuits directly, and it is this extra layer that makes it richer than simple instinct.

It is often said that design is domain specific[7] and that this means that one

[7]This proposition would seem to be in the same spirit as the idea that reasoning, or, in contemporary terms, critical thinking, only occurs in context and can therefore "only be taught as part of a specific subject and never in isolation" (McPeck 1981, p. 158). Intuitively this seems unlikely, for an implicit assumption of any education system is that transfer across subject domains will occur. " For a very long time, it has been common practice for mathematics departments to teach the quantitative models of reasoning (e.g., algebra, calculus, etc.) assumed

can't teach programming in general, it needs to be presented always in the context of a specific domain - network programming, or systems programming, for example. Even if it is true that design is domain specific, I can't see how that makes any difference to the teaching of programming. The point about 'design' is that you are teaching it whatever you do - even if you think that you're not. Any program is 'designed'. You might think that you are just 'hacking code', but, in fact, your program still has the elements that constitute 'a design'. It's a consciously, intentionally produced artefact. You might not have explicitly 'designed' it in a conscious sense, but whatever, it has a 'design' and you put it there, it didn't just happen.

The point about hacking is not that it doesn't produce a designed artefact, but that it produces a design that is not based on understanding, "the programmer's understanding of the problem to which he is trying to program a solution usually develops only with the program" (Brady 1977, p. 13), that is, after the fact. It is important to notice that this is *not* a problem in itself given a high degree of internalised programming skill. Like the musician who *improvises* a piece of music, it is *experience*, lots of it, which allows the design (composition) and implementation (playing) to occur simultaneously in the expert's performance. But the level of *understanding* that is required for this to be successful is far beyond the novice performer in either field, it comes only after years of practice.

In fact, it is this conjunction between experience and programming skill that has caused the two contrasting meanings of the term 'to hack'. Initially the term 'hacker' was used to denote a highly proficient programmer who could 'hack' up a code solution in no time. The term, in this case, probably derived from the expert's proficiency in reusing code - previously written code was 'hacked' (in the sense of being altered) to produce the new program. But this ability was based on a highly developed understanding arising from years of "blood, sweat and tears" that meant that the design process that was most certainly occurring was not visible to the admiring observers. The trouble came about when others who did not posses the same degree of understanding tried to emulate the expert's performance with generally disastrous results. In this case it was the solution that ended up being 'hacked' - but 'hacked' in the sense that a piece of wood is 'hacked' to useless pieces by an inexpert carpenter trying to design on the fly.

So whatever you might think that you are doing when you teach people programming you are implicitly teaching them to 'design' programs. I don't see how you can avoid it. All that the pattern idea says in this regard is that if I'm teaching design then it is better to do it upfront, to make it explicit. This surely is the thrust of the pedagogical patterns movement. Because we don't tell the students

necessary for acquiring and applying concepts in other disciplines. We do not assume that the disciplines utilizing these models (i.e., chemistry, physics, psychology, etc.) should themselves be responsible for teaching algebra, for example. Indeed, we find it efficient to require mathematics courses as prerequisites to the study of a variety of complex subject matter domains" (Nummedal 1987).

that we are teaching them, or rather, trying to teach them, design, they miss many important issues completely. This is why they struggle with it. We make them 'hackers' - programmers who *design by default* while coding - without the necessary background to be able to do this successfully. Pedagogically, it must be advantageous to explicitly teach novices to design their programs, before they begin coding, because, at the very least, it will cause *some* attention to be given consciously to thinking about the problem and its solution at a conceptual level. In a sense, teaching design explicitly is merely implementing Bloom's taxonomy, one is addressing analysis before synthesis.

The difficulty with the basic premise under discussion is that if design is domain specific then this has to be true in *every* field not just programming, so the statement comes down to 'we can *never* teach design *except* in the context of a particular domain'. But what then is a domain? It's probably not possible to go anywhere else but to a definition that says, in effect, that 'a domain is a collection of related systems'. In our case, patterns are problem based so I guess that a more specific definition for us is - a set of related problems and the mechanisms for solving them. But aren't we then just talking about the level, or scope, of concepts when we use the term 'domain'?

So how do we decide that problems are related? This is purely a pragmatic issue. What is really being said is that there are a number of concepts that all contribute to a certain level of order that we are interested in achieving. What we are doing is building a system in the sense that a system is a collection of entities and concepts about some sort of organisation. The problems arise in creating, extending or maintaining the conceptual system. That is, they are system level organisational problems. The difficulty of the domain idea is that it is merely another system, but a system that can contain many systems. So deciding what constitutes a domain is simply a matter of choice, it is whatever set of entities, including systems, that it proves useful to regard as constituting a larger system to be known as a domain. The thrust of Alexander's "Nature of Order" is that moral order is reflected in everything. In a sense, everything ultimately relates to the same overall order. But when dealing with particular issues it just adds complexity to relate them to everything, so the first thing we do is to simplify everything by breaking it down into a number of smaller categories. Ultimately, a category is merely a convenience. It is helpful to think of a set of related ideas in this way, so we do it. And deciding if a particular category is a domain is similarly a matter of convenience.

What the contention boils down to, then, is the statement that 'we can NEVER teach design' because domains are ultimately just conveniences. The only way to escape this is to modify the contention to something like 'we can NEVER teach design EXCEPT by recognising that every design principle that we teach has a context'. But this is the fundamental crux of the pedagogical pattern language idea. Instead of pretending that programming is design-context-free we base our teaching program on the context of concepts and, moreover, use context as the driving force of the process of design.

The point about pattern languages is that they exist at all levels. You can think of them as existing at the level of individual projects within a category of systems, at system level within the category, and at the overall category level. All these levels of categorisation are merely convenient ways of relating a set of concepts. There is nothing fixed or permanent about any category. It is merely a way of expressing a level of order. Any single person cannot know everything, so we break it down into broad areas of related ideas such as Physics, Chemistry, History, and so on. In turn each of these broad categories quickly grows beyond the power of one mind to encompass so we specialise even further European History, American History, Medieval History, etc. Because a pattern language is a reflection of order it can be useful at all levels. A pattern language is built whenever order is being created or maintained. This is why Alexander always stresses the idea of process. If you are thinking about doing something, you are considering order in some sense. The level or scope of the activity that is being considered determines the scope of the pattern language that you need to use. If it makes sense to think of a set of related concepts as a category, or some other sort of level of order, like project or system, then it makes sense to think of a pattern language at that level.

All the pedagogical pattern language idea is saying is that 'basic programming' is the domain for our purposes. Whatever the task to be accomplished is, the first problem we face is that of writing a program to carry it out. This is saying nothing else but that the first or topmost pattern in our pattern language, whether this is made explicit or not, is `PROGRAM`, that is, consider the implications of carrying out the specified task by means of a computer program. In order to program a specific task we need to understand it in terms of the facilities that a computer provides. The primary order involved in the field is the use of computers to automate tasks. This is done by programming a general-purpose machine, the computer, to carry out a specific task. So the first requirement of any person designing programs in any specific domain is that they understand the facilities provided by the toolkit they are using, the programming language. Any program design in any specific domain will require this base level of design nous. So our approach, at least, acknowledges design and teaches that it is grounded in context.

Any task arises in a particular context in a domain, for example, accounting, which has many features, already existent, that are used in solving problems, completing tasks, in the domain. It has, in other words, a pattern language specific to that domain. The problem to be solved exists in that domain and it will be *specified* in the language of that domain. Importing into the domain in which it arises, the initial pattern, `PROGRAM`, is therefore a way of considering solving the initial problem by automating the task - doing it on a computer. It is the power of the pattern language idea that doing this imports the language attached to `PROGRAM` so that it is now clear that the way forward, if automation is deemed appropriate as a solution to the initial problem, is to work through the rest of the language linked to `PROGRAM`.

This is important. In fact it is the *breakthrough* idea in terms of applying Alexander's thinking to programming practice. Suddenly, a whole new way of thinking about the initial problem is opened up - with a whole new set of features and ways to proceed. Applying a pattern *always* does this. It transforms the context of the problem into the context of the pattern. We go into this in further detail in Chapter 6, but in outline context is everything in establishing *meaning* and it is only on the basis of *meaning* that problems can be understood and solved.

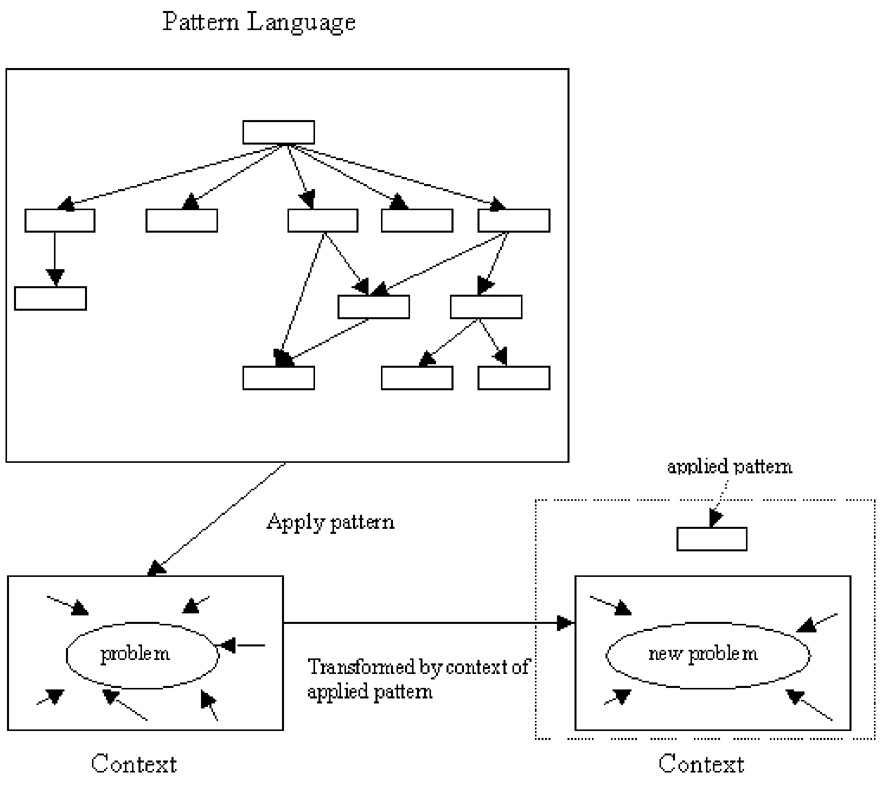


Figure 4.1. The applied pattern brings its context with it.

So the thrust here is that the pattern approach does not fundamentally challenge the premise. Yes, design is domain specific. But any domain is just a category. It is whatever is convenient for your purposes. Our purpose is to teach people programming. Any program embodies a design, implicitly or explicitly. We claim that making the design process explicit is advantageous in writing any program not just a program in a particular context. So yes, we admit that we cannot teach the domain specific design principles for every single context in which a program will be written, but there are some program design issues that are relevant in EVERY specific domain in which programming is done. Thus, for our purposes, we define our domain as 'basic programming', and present design in that context. So the contention that 'design is domain specific' boils down to the statement that programming can NEVER be taught EXCEPT through pattern languages.

1. every program constitutes a design
2. design is domain specific
3. a domain is just a convenient set of contexts
4. a pattern language is a convenient set of contexts (problems in contexts)

   It follows from 3 and 4 that
5. every domain has a pattern language

   Therefore, because
6. the 'fundamental process' is design - "There is something *essentially* dynamic about order. ... Indeed *the nature of order is interwoven in its fundamental character with the nature of the processes which create the order.* (Alexander 2002*a*, p. 2)

   every design involves the use of a pattern language, like it or not.

   But
7. teaching programming involves writing programs and every program constitutes a design (see point 1)

   Therefore
8. we are teaching the 'fundamental process'.

We are teaching a pattern language.

Giving this fact due recognition is the main point of this thesis and as Alan Perlis once said: "A language that doesn't affect the way you think about programming, is not worth knowing" (Perlis 1982). Programming languages are for writing code, not designing programs, so it is not surprising that we have experienced great difficulties in teaching people when the languages we use are not really about thinking about programming. The creative aspect of programming occurs at the *design* level not the *code* level. There is nothing that is particularly new in all this. Design is fundamentally creative, it is simply the nature of the beast. As an activity it only occurs because purpose or need overruns current circumstance, we respond to a problem by 'designing' a solution. Creativity and design are responses to uncertainty, and in this sense are virtually synonymous. Were it possible to *know* every fact about a situation then it would be possible to deal with it using formal logic. But the life of a 'social animal' involves factors that are almost infinitely variable so we simply have to rely on patterns of probability.

> Nature's patterns are "emergent phenomena." They emerge from an ocean of complexity like Botticelli's Venus from her half shell - unheralded, transcending their origins. They are not *direct* consequences of the deep simplicities of natural law; those laws operate at the wrong level for that. They are without doubt *indirect* consequences of the deep simplicities of nature, but the route from cause to effect becomes so complicated that no one can follow it [Emphasis in original].
>
> (Stewart 1995, p. 146)

The basis of this project is the paradox between the simplicity of the programming environment and the massive difficulty that many people encounter in trying to learn to program. Paradox, as a misalignment of meanings, always points to a philosophical issue, because philosophy is the systematic study of meanings (Langer 1962, p. 97).

At the base of our pedagogical problem lies a conceptual dissonance of some kind. This is sharply demonstrated in the various social anomalies, like the disinclination of women to enter fields of endeavour like Computer Science that are perceived to too heavily involve mathematic or engineering methodology. Researchers have been investigating cognitive differences between people in different areas for some time.

> One of the areas that they have been researching with respect to gender is the area of cognitive differences between males and females. The results have been reproduced several times by different researchers in different areas of the United States, and so it seems fairly sure that there is a definite difference in the way that boys and girls think in the math classroom. This cognitive difference has definite repercussions to those students' achievement as they progress through the system. Do women really lack the ability to do math? Certainly not. However, gender differences in mathematical achievement in the classroom are a very real phenomenon. This is due to several factors, once of which is the difference in cognitive style between males and females at a young age. These cognitive differences are reflected in the different achievement that males and females reach using standard, traditional assessment methods. In essence, boys and girls think and solve problems in different ways, and standard assessment techniques do not address the effects of these cognitive differences.
>
> (Mahood & Richards 2000)

But these differences in the way that people see the world is even more powerfully illustrated by the stories told by many of their personal experiences of learning to program. Turkle and Papert discuss a couple of these histories in several articles published in the early 1990s.

> Consider Lisa, 18, a first-year Harvard student in an introductory programming course. Lisa had feared that she would find the course

> difficult because she is a poet, "good with words, not numbers." But after years of scorning teachers who had insisted that mathematics is a language, the computer has made Lisa ready to reconsider the proposition, and with it her characterization of herself as someone "bad at math." Lisa started well, surprised to find herself easily in command of the course material. But as the term progressed she reluctantly decided that she "had to be a different kind of person with the machine." She could no longer resist a pressure to think in ways that were not her own. She was in trouble, but her difficulty expressed a strength, not a weakness. Her growing sense of alienation did not stem from an inability to cope with programming but from her ability to handle it in a way that came into conflict with the computer culture she had entered.
>
> (Turkle & Papert 1992)

What is being expressed here is a clear, almost shocking, dissonance between the way that the subject *is*, and the way that she feels programming, or at least the culture that surrounds it, is forcing her to be. As educators we have all come across the phenomena of *really* intelligent people, people who go on to excel in other fields, often, curiously enough, biology, who fail to cope with learning to program. It is too easy to see this as a personal disinclination for the 'hard' sciences, but, I think, the basis of the problem is rather the *inclination* of those who teach Computer Science to view it as a 'hard' science. In fact, if by 'hard' we mean the rigidity of mathematical and logical thinking then we are wrong, at least as far as programming is concerned. Designing a solution to a problem is almost totally an act of developing a theory about the problem space, just like thinking in 'softer' sciences, like biology in fact. When one is theorising, one is partaking in a creative activity and in this mode one's thinking needs to be as free and flexible as possible. If students could be given more of this flavour of the programming realm, and the corresponding fact that successfully translating your design into code constitutes a formal proof of your theory, then maybe programming would be regarded in a more favourable light by those who are made to feel 'alien' in the Computer Science culture by the mathematical and logical rigidity of coding.

This paradox between the basic simplicity of the programming environment in terms of the very small number of concepts it encompasses, and the perceived mathematical and engineering rigor of the conceptual field as a whole, is the probable source of the difficulties in teaching programming. Resolving conceptual difficulties is not simply a matter of science, usually the basic facts are well known, what is missing is 'understanding'. Unfortunately scientists, and in particular, computer scientists, are generally impatient with philosophical discussions[8] and

[8]This is part of a wider problem, namely the fact that practice has outpaced theory in many fields. It is clear that as a society we have not yet come to terms with the philosophical implications of advances in medical, genetic and military sciences either. "The current doping agony [on the sports field] ... is a kind of very confused referendum on the future of human

this has led to an almost complete dearth of consideration of what Edward Yourdon calls the "*philosophies* of programming" [emphasis in original] (Yourdon 1975, p. 2). We believe that most of the difficulties in this area stem from the basic misconception that programming is writing code when, in fact, "representation *is* the essence of programming" [emphasis in original] (Brooks 1982, p. 103), not coding, and representation is an issue of much wider compass than computer science.

To use the military analogy, coding is tactical level thinking not strategy, and no war was ever won in the absence of a comprehensive *design* for victory, that is, a strategy. In fact the overemphasis of the tactical is notorious in military theory as the genesis of major disasters like the second battle of the Somme or the whole Gallipoli campaign in the first war, and Hitler's hands-on conduct of the second. Although the thinking behind these policies was considered strategic, it is clear that the main effect was tactical paralysis, which is surely always bad strategy. Hitler's general order for no retreat under *any* circumstances contradicts the essential need for strategic flexibility, it precludes the possibility of regrouping in the face of a setback. The point about the decision to stand and fight or retreat is that it is a tactical not a strategic one. Sometimes standing ground is the right thing to do but more often than not it isn't, so to issue such an order as part of overall strategy is to hamstring tactics and therefore to blind strategic thinking. In programming language terms these are local variables rather than global ones. But the really surprising thing is how often we seem to have to learn, and relearn, the absolute primacy of having an holistic appreciation of any problematic situation. Details are ultimately important *only* in terms of the whole, allowed to dominate one's thinking they dictate overall direction by default, rather than being considered in the light of an overall view. Before we can significantly improve the pedagogical situation in programming we have to better understand programming as a mental activity, that is, how to more closely fit the task to the way that the mind works (a philosophical undertaking), rather than simply trying to modify the way that the mind works to fit the way that *we*, as computer specialists, see programming.

enhancement" (John Hoberman quoted in (Garreau) 2005), for example. Already there are research programs aimed at "modifying the metabolisms of soldiers so as to allow them to function without sleep or even food for as much as a week ... [and] for shorter periods ... without oxygen"(Garreau 2005), yet we are still mired in the drugs in sport debate. The reason that the convergence of medical, genetic and military science is being debated in this indirect and ad hoc way is precisely because, as a society, we don't have the philosophical, or social, mental infrastructure on which to base the discussion. In short there needs to be a more interdisciplinary approach to look at new ideas in a wider context generally, and this is starting to be recognised - see the April 14 2005 draft report of "The President's Information Technology Advisory Panel" which recommended the restructuring of universities and federal agencies to promote multidisciplinary research. (reported in (Chronicle of Higher Education) April 15 2005) . Of course, given the context of the panel, one can suspect that this push is driven more by the necessity to justify government policy in the minds of the general public than to genuinely inform a real debate.

Paradoxically, becoming an expert involves increasing skill by decreasing the potentials. Faced with a problem the beginner is overwhelmed by the possibilities, there are too many choices, too many avenues to explore. Creativity is more than just coming up with the novel. Stringing together, willy nilly, any sequence of musical notes does not make a tune. A tune requires, at the very least, musical structure, so musical creativity needs to be directed towards forming structure. Before one can be an expert composer, one must develop a musician's mind. Everybody starts with a brain but no mind. Mind is the product of experience in a way that the brain is not - or not, at least, in any significant measure. The relationship between the two is not 'mind is brain', it is 'mind requires brain'. Human development, as described by Piaget and others, is simply the unfolding development of mind. What it results in is the general mind that all adult humans need to 'get on' in their world. They have become experts in the social milieu and physical environment into which they were born. In a sense they have created themselves and their world, their mind, in a dynamic relationship with their environment.

Mind is not simply brain - it requires a brain in which to be based in a physical sense, like a life needs a body - but to think of mind in terms of its physical manifestation is just too restrictive. A concept is mind stuff, not brain stuff. As one of the pioneers of neuro-surgery, Wilder Penfield, explains, it is not possible to conjure mind-mechanism by means of electrical stimulation of part of the brain.

> I have been alert to the importance of studying the results of electrode stimulation of the brain of a conscious man, and have recorded the results as accurately and completely as I could. The electrode can present to the patient various crude sensations. It can cause him to turn head and eyes, or to move the limbs, or to vocalize and swallow. It may recall vivid re-experience of the past, or present to him an illusion that present experience is familiar, or that the things he sees are growing large and coming near. But he remains aloof. He passes judgment on it all. He says "things seem familiar," not "I have been through this before." He says, "things are growing larger," but he does not move for fear of being run over. If the electrode moves his right hand, he does not say, "I wanted to move it." He may, however, reach over with the left hand and oppose his action. There is no place in the cerebral cortex where electrical stimulation will cause a patient to believe or to decide.
>
> (Penfield 1975, pp. 76-7)

Expertise is based on the mind, not the brain directly. If one had to replay every event memory that contributes to one's understanding of a particular concept every time one needed to use it, then nothing would ever get done. "People are good at recognition, but tend to be poor at recall" (Liddle 1996, p. 21). Concepts are more than just memories, brain stuff, they are mind stuff, patterns.

The notion of abstraction is not just conceptual in nature, it is actual - an idea is not neuronal, it 'really is' abstract, and the mind is the total collection of the abstractions that enable it to function as a unit, a 'whole'. So, just as becoming an independent human is to create, from the raw material of neuronal structure, the mind of a human in a particular environmental and cultural context, becoming an expert in a particular field of endeavour is similarly to create the mind of an expert of that field. But this time you start with a mind already developed, not just a brain still awaiting the development of mind, in other words, a brain already habituated to dealing in pattern language.

4.5 The Programming Experience

It is in this sense that we argue that there is no "special" characteristic or "trait" involved in the task of programming, save the ability to use a language. In a sense, the fact of having learned to speak demonstrates both the possession of the required facilities, and the ability to develop them fully. Programming is basically the task of "so analysing and paraphrasing a problem as to cause its solution to organize itself into steps which a machine can be made to take" (Quine 1966, p. 40). One could define the use of natural language in similar terms, that is, as the task of "so analysing and paraphrasing a situation as to cause its expression in words to organize itself into steps which the language can be made to take". However, this correspondence between natural language and programming occurs, not because, as is usually supposed, programming is just another language, but because both the use of a natural language, and programming, are developments of thinking skill - they both descend from the same ancestor, rather than one from the other. The implications of this are quite profound because it would seem from the history of programming that the language metaphor has been taken too literally - we have treated programming as if it were a direct descendant of natural language rather than as a close cousin, missing the fact that it has more in common with thinking than it has with speech or writing.

Natural language stands in the same relationship in the human communicative system as the machine does in the computational system (although one has to be careful with this idea as it leads to the classic "information processing" confusion[9]). This similarity is demonstrated by the fact that in the following equation the language component is interchangeable.

$$\text{human - language} \longrightarrow \text{communication}$$

Language, here, can be any device that processes linguistic relationships - sounds, written symbols, or even hand or finger/touch signs - just as a machine in the

[9]Unfortunately, the phrase 'information processing' has come to carry a connotation of computing which is entirely inappropriate to describe human mental processes (Devlin 1995, p. 6). Strictly speaking, computers do not 'process information' at all, they manipulate symbols that, because they have no 'meaning' for the machine, only carry 'information' as such for the human operators.

next equation can be any device capable of processing logical relationships.

$$\text{human - machine} \longrightarrow \text{computation}$$

In the case of the machine, it can take forms based on beads, digits on paper, and so on, as well as the more conventional voltages in electronic components.

Strictly speaking, a natural language is a system to enable communication just as a computer is a system to enable computation, and both require ‘programming’ to accomplish their ends. In this regard the so-called ‘natural language’ is as much a human artefact as any ‘physical’ machine, but we miss this fact because of its lack of material form[10]. Both are tools that are used by humans to manipulate the relations between symbols to generate new meaning, so setting up the linguistic system to produce communicative meaning is as much the task of ‘programming’ as setting up the computing system to produce computational meaning. Another way of saying this is that programming a computer to compute is equivalent to thinking of what to say. So the two equations mentioned above are, in reality, just different forms of the following relationship, where ‘system’ can be communicative or computational. After all, the ‘mind’ does both sorts of processing, it can fit into ‘machine’ in the second equation just as it can fit into ‘language’ in the first.

$$\begin{tikzcd}\text{Human} \arrow[r, "\text{thinking}"] & \boxed{\text{system}} \arrow[r, "\text{processing}"] & \text{meaning}\end{tikzcd}$$

Figure 4.2. The general form of the communication/computation relationship

Thinking of the situation like this should help us see the source of the difficulties that people have with programming a computer. If thinking about what to say is, generally, so easy, why should programming a computer be so problematic? Something about the way we program a computer makes it much more difficult than programming our linguistic system. We feel that the source of the problem is the tight correlation that has been made between the computing system and the method of programming it. Because we have failed to see that language is equivalent to machine in the two systems, we have stood the programming process in the wrong place in the computational system. By this we mean that we have tied the language used in computer programming too tightly to the logic of the machine. The real correspondence is between thinking and programming, so if the ‘language’ used in programming is restricted to logic this makes it very unlike the freeflowing form that is thinking. Thinking is not even restricted to just those symbols that exist in the linguistic system, but makes use of concepts that are more like images (Bohm 1980, p. xiv), and relationships that are not strictly grammatical[11].

[10]Of course this is not strictly true, the material manifestation of the language system, is the brain, or at least those parts of the brain that give rise to the ‘mind’. The brain - mind - language relationship is discussed further in this work.

[11]This is true even of speech, where some elements of written grammar, e.g. the use of

What this implies is that there are probably several levels involved in the process of thinking, each level being more 'logically organised', or perhaps one should say 'linguistically organised', than the one before it. It seems that there is some evidence for a preliminary stage to conscious thought (Luriia 1973) that has little or no explicit organization. The task of the brain is to build this diffuse mass into the structural form that we call 'mind'. Narciss Ach called this pre-organizational mass 'imageless knowing' (Association GREX n.d., p. 14). What we experience as the conscious and symbolic organization of everyday thought is derived from this initial state of 'imageless knowing', which therefore underlies imaginal experience, conscious thought, language usage, and eventually behaviour. In a sense then, that entity that we know as mind is itself a product of creativity - it derives "from an indefinite non-organized state that is prior to consciousness to one of increasing distinctiveness which achieves form and organization characteristic of thought and consciousness" (Joseph 1980).

Linguistic form, or communication, is not only the end product of this development process, but contributes to it. In fact each level of thought flows from and feeds back into the preceding levels so that it becomes virtually impossible to consciously disentangle them. It can almost seem as if one is 'talking' to oneself which raises some interesting philosophical issues.

> Directed, reflexive or spontaneous, the verbal thoughts always unfold before an observer and are heard within the minds ear. It is a series of pseudo-auditory transactions which are experienced as well as produced sometimes as a purposeful means of explanation. Thinking sometimes is experienced as a form of self-explanation through which ideas, impulses, desires, or thing-in-the-world may be understood, comprehended and possibly acted upon. Paradoxically, it is often a process by which one explains things to oneself. Indeed, as a means of deduction or explanation, and as a form of internal language, it is almost as if one is talking to oneself inside one's head.
> Nevertheless, the fact that one acts as both audience and orator, raises a curious question: "who is explaining what to whom?" A functional duality and in fact a functional multiplicity is thus implied in the production and reception of thought.
>
> (Joseph n.d.)

It would be, of course, altogether two simplistic to equate this apparent duality of mind to different areas of the brain, nevertheless the principle of 'form following function' probably does apply to the neural structure. There is clear evidence of functional specialization in the brain, that "the differentiation of neural structures

sentences, can be largely discarded. A lot of the rules of grammar (syntax) are concerned with structuring written language and are therefore applied only loosely to spoken language and hardly at all to thought. After all, if you are thinking, you are the only person who needs to understand what you are thinking. Grammar is a function of the need to communicate, not strictly the need to understand.

may be shaped and patterned through the ontogenetic and phylogenetic evolution of consciousness, and physically mirror the two types of consciousness necessary for adaptation" (Cooperstein n.d.). This development does not just occur through the process of biological evolution, the development of the individual builds on the basic structure provided by the expression of the genetic inheritance.

> Metaphorically, consider the fetal brain a gelatinous mass of undifferentiated, unspecialized cellular materials. Further, let us assume that embedded in this matrix are myriad hereditary neural pathways, all possible avenues for the transmission of energy to specific areas. The priorities of the pathways chosen, and the areas they service, are assigned by the genetic code, providing the anlage for maximal survival potential in the majority of our species. A remarkable consistency in laying down these potentials for later differentiation encompasses certain "universals" for adaptation, prepotencies for collective perception and action, or, what some have called instincts. ... In this way, the primitive nervous system is guided to establish prepotencies for adaption as rudimentary, pre-systemic configurations already subjected to activation concurrent with their appearance. Many theorists have identified as archaic forms or potentials (e.g., Jung's archetypes, Kant's priori categories, Jaynes' aptic structures, and Arieti's endocepts) [that] may well be common recognition of the biological substrate as it evolves into being, arising from humankind's common history of adjustment to this planet, now part of a genetic record encoded in which is the past, present and, perhaps, the future of humanity's adaptation.
>
> (Cooperstein n.d.)

But the main issue here is process. At a fundamental level all process is creative, a new state has developed out of a previous state, "wherever development occurs, it proceeds from a state of relative globality and lack of differentiation to a state of increasing differentiation, articulation, and hierarchic integration" (Werner 1957, p. 127). Thought is thus inherently a form of creation and, in the end it is probably impossible to say which is primary in the development of mind, a process, just as it is difficult to define the brain-mind relationship, which is also the product of a process. Any thinking organism is therefore creative at a fundamental level. Maybe many of the nature-nurture type controversies derive from this fundamental conjunction of thought and creativity. In a sense, all humans have an 'aptitude' for thinking, so many of the apparent aptitudes for particular skills that involve creativity are probably just expressions of this general aptitude. What looks like an aptitude for programming is, in essence, the aptitude for thought directed by an interest in programming. If this is true then motivation is the key to learning how to program, in the end "interest" might be more important than any cognitive "aptitude", might indeed be the source of the "aptitude" phenomenon.

> That's why ... the work of Silvan Tomkins [is] just so useful because he said, 'you know, with affects anything can matter and without affects nothing does'. Like if I'm in a sort of curious, interested and excited affective state then I might find absolutely everything about the world fascinating and I'll be noticing the branches on the trees and, you know, the shape of the furrows and fields and everything. In other words it renders salient in so many different parts of the world. If you're in a sort of deflated depressed or low affective state where there's not really any affects, sort of moving and inspiring you, the world feels quite flat and grey and not very much matters to you at all.
>
> (McIllwain 2005)

If affects are primary then one of the main factors in helping someone learn anything is the attempt to inculcate and maintain interest in whatever it is that is being learned. So a pedagogy that produces great difficulty in students is something that needs to be avoided because perceived difficulty is bound to result in flagging interest in many people. Unfortunately, with the ascendancy of cognitive psychology, affect has come to be seen almost as "a regrettable flaw in an otherwise perfect cognitive machine" (Scherer 1984, p. 293) presumably because it does not fit in with the brain-as-computer model. However, it is clear from any sensible contemplation of the human condition that emphasizing only affect or only cognition is just silly. As Vygotskii said, the separation of affect from cognition "is a major weakness of traditional psychology since it makes the thought process appear as an autonomous flow of 'thoughts thinking themselves,' segregated from the fullness of life, from the personal needs and interests, the inclinations and impulses, of the thinker. Such segregated thought must be viewed either as a meaningless epiphenomenon incapable of changing anything in the life or conduct of a person or else as some kind of primeval force exerting an influence on personal life in an inexplicable, mysterious way" (Vygotskii 1962, p. 10). In order to understand the whole person, rather than just discrete phenomena associated with them, cognition must be viewed in concert with affect.

Chapter 5

Language as Understanding

Man looks at his world through transparent patterns or templates which he creates and then attempts to fit over the realities of which the world is composed. The fit is not always very good. Yet without such patterns the world appears to be such an undifferentiated homogeneity that man is unable to make any sense out of it. Even a poor fit is more helpful to him than nothing at all.

(George Kelly, 1963 quoted in (Britton) 1970, p. 17)

With the exception of music, we have been trained to think of patterns as fixed affairs. It's easier and lazier that way, but, of course, all nonsense. The right way to begin to think of the pattern which connects is to think of a dance of interacting parts, pegged down by various sorts of limits.

Gregory Bateson - Cultural Anthropologist

5.1 Programming and Problem Solving

Most of the difficulty with programming seems to lie in the general problem solving aspect rather than the specific details of the programming language. That is, before you can write a program that solves a problem expressed in normal language you first have to solve it in general terms. There is a stage that involves coming to an understanding of the problem and its solution in conceptual terms that precedes solving it in programming terms. The programming stage, in its narrowest sense, is merely the process of translating the conceptual solution into a sequence of programming language constructs (Deek & Friedman 2002) and there is little or no creativity at this level. Specifying a problem to which a computer program solution is required involves a natural language description. Therefore the largest and most difficult part of the programming task is deriving the required conceptual understanding from the natural language specification to produce the solution at the conceptual level rather than the code level, and what

Alan Perlis said about programming levels, "a programming language is low level when its programs require attention to the irrelevant" (Perlis 1982) is true about all programming languages in relation to conceptual understanding because they express code not concept and the code is irrelevant in conceptual terms.

> One of the main challenges is that of transforming an ill-stated problem to one acceptable to the computer. ... The programmer has to select a methodology by which he can take an ill-stated problem and restate it in such a way that it can be developed into a set of procedures and expressed in a language that is acceptable and understood by the machine. Once a problem is well-stated, communication becomes relatively simple.
>
> (Martin & Badre 1977)

A pattern language is primarily a way of understanding complexity. Any system is a set of entities with some level of organisation. If a system is completely disordered then it can be described statistically using the law of large numbers (Brownian motion, for example). If it is in a completely ordered state then it can be described using traditional deterministic methods (crystalline structure). But most systems are not in either of these two states. These two states represent the ends of the organisational continuum. In most real world systems there is some level of organisation that is more than complete disorder, but less than complete order. Patterns are a means of discerning what level of organisation there is in a particular system, they are telling you about existing order. They are important in organisational terms because of this.

Understanding a domain is more than just knowledge of the elements in that domain, it is mostly about structure. When a person is said to have an understanding of a domain it means that she can 'stand' the elements of which it consists into a structure that is both coherent and correct. It is based on the concepts 'standing under' each other in a structural (hierarchical) form that reflects the relationships between the real elements, the order of the system. It is only with this appreciation of structure, 'understanding' or concepts 'standing under' each other, that the functioning of a system can be approached in any sensible manner.

A software pattern is an expression of some aspect of the existing order of a system. It says, in effect, that in this context, this solution is known to solve this problem. Order in a system can be increased by understanding what currently works in it, (the patterns), and using that knowledge to develop new processes and improve old ones. Software patterns tell you what works in a programming context, the language structure, the hierarchical relationships between them, provides the means of combining them to increase system organisation. In other words a pattern language for a programming domain provides a methodology by which an ill-stated problem "can be developed into a set of procedures and expressed in a language that is acceptable and understood by the machine"(Martin & Badre 1977).

This work argues that programming is based on the use of a pattern language whether this is conscious or not. We say that it is plausible to argue this because all human activity is so based. Language has often been seen as the core attribute of the human condition and this is true, but the normal usage of the word language obscures the fact that it is language at a lower level than the usual meaning of the word, language-as-communication. The primary function of language in terms of being human is not communication but in providing the means of understanding, the means by which we make sense of the world. In essence, pattern and language skill form a functioning isomorphism, called 'meaning', which allows us to understand complex systems. Language, in the sense in which we are using the term here, is the system by which humans organise their thinking about experience, a semantic system. "The real structure of language lies in the relationships between words - the semantic connections. The semantic network - which connects the word 'fire' with 'burn,' 'red,' and 'passion' - is the real stuff of language" (Alexander quoted in (Grabow) 1983, p. 50).

However this idea pushes back, in terms of human development, the concept of language. "That language in which information is communicated [in the brain]... neither needs to be nor is apt to be built on the plan of those languages men use toward one another" (McCullock 1965, p. 56). We have tended to use the word language to denote that capacity that surrounds the conscious use of words, that is, it is seen primarily as a means of human communication rather than the basis of the way that an individual understands the world. This normal usage of the term exaggerates the communicative aspect (Edward Sapir quoted in (Langer) 1976, pp. 109-10) and obscures the primacy of mental function over the communicative aspect of language, and the two threads require some untangling. One might even say, as Steven Pinker suggests, that thought is based on metaphysics ("concepts and relations") rather than language in its communicative sense, and hence the appropriateness of object-oriented thinking in the design of programs.

> Many cognitive scientists (including me) have concluded from their research on language that a handful of concepts about places, paths, motions, agency, and causation underlie the literal or figurative meanings of tens of thousands of words and constructions, not only in English but in every other language that has been studied. ... These concepts and relations appear to be the vocabulary and syntax of mentalese, the language of thought. ... And the discovery that the elements of mentalese are based on places and projectiles has implications for both where the language of thought came from and how we put it to use in modern times.
>
> (Pinker 1997)

What is important, here, is the primacy of *thought* in terms of combining, and even, indeed, having, ideas, and the famous case of Helen Keller, blind and deaf from very early childhood, well illuminates the almost miraculous nature of language-as-understanding. For her the sequence of development was, to all intents and purposes, reversed from that experienced by normally sensed people. She had learned to communicate at a basic level using touch signs but the real power of thinking, "understanding," awaited the arrival of "meaning." It is worth quoting at some length Susanne Langer's description of the dawning of "meaning" for the young Helen.

> There is a famous passage in the autobiography of Helen Keller, in which this remarkable woman describes the dawn of Language upon her mind. Of course she had used signs before, formed associations, learned to expect things and identify people or places; but there was a great day when all sign-meaning was eclipsed and dwarfed by the discovery that a certain datum in her limited sense-world had a denotation, that a particular act of her fingers constituted a word. This event had required a long preparation; the child had learned many finger acts, but they were as yet a meaningless play. Then, one day, her teacher took her out to walk - and there the great advent of Language occurred.
>
> "She brought me my hat," the memoir reads, "and I knew I was going out into the warm sunshine. This thought, if a wordless sensation may be called a thought, made me hop and skip with pleasure.
>
> "We walked down the path to the well-house, attracted by the fragrance of the honeysuckle with which it was covered. Some one was drawing water and my teacher placed my hand under the spout. As the cool stream gushed over my hand she spelled into the other the word water, first slowly, then rapidly. I stood still, my whole attention fixed upon the motion of her fingers. Suddenly I felt a misty consciousness as of something forgotten - a thrill of returning thought; and somehow the mystery of language was revealed to me. I knew then that w-a-t-e-r meant the wonderful cool something that was flowing over my hand. That living word awakened my soul, gave it light, hope, joy, set it free! There were barriers still, it is true, but barriers that in time could be swept away.
>
> "I left the well-house eager to learn. Everything had a name, and each name gave birth to a new thought. As we returned to the house every object which I touched seemed to quiver with life. That was because I saw everything with the strange, new sight that had come to me.
>
> This passage is the best affidavit we could hope to find for the genuine difference between sign and symbol. The sign is something to act upon, or a means to command action; the symbol is an instrument of thought. Note how Miss Keller qualifies the mental process just preceding her discovery of words - "This thought, *if a wordless sensation may be called a thought*" [Langer's emphasis]. Real thinking is possible only in the light of genuine language, no matter how limited, how primitive; in her case, it became possible with the discovery that "w-a-t-e-r" was not necessarily a sign that water was wanted or

> expected, but was the name of this substance, by which it could be mentioned, conceived, remembered.
>
> (Langer 1976, pp. 62-3)

For the child with normal sense facilities, the connection between symbol and concept is much easier to make. But, more than that, the essential generativity of language is assimilated with no formal knowledge of grammar, showing the ordinary linguistic creativity of everyday speech derives from the *semantic* connectivity between concepts (pattern language), not the formal syntactic language structure. In fact, syntax is acquired almost entirely (there is some corrective input from adults) as a 'side effect' of the expression of meaning, that is semantic creativity. Despite being massively incompetent at most adult capabilities, for example, being "flummoxed by no-brainer tasks like sorting beads in order of size" (Pinker 1994, p. 276), the average three-year-old "is a grammatical genius - master of most constructions, obeying rules far more often than flouting them, respecting language universals, erring in sensible, adultlike ways, and avoiding many kinds of error altogether" (Pinker 1994, p. 276). And all this, let it be remembered, with **NO** overt instruction in the syntax, the logic, in effect, of the language, this is the assimilation of patterns in the child's hearing environment, and it is driven by the need to *understand* the world, by the semantic impulse.

What Helen Keller's case illustrates is the essential symbiosis of language and mental organisation. Although she had learned many "finger acts" (words), they were essentially meaningless ("wordless sensation") until the ability to organise her thinking in a language-like manner arose. Mental organisation is best seen as the primary formulation of language, and it derives from sensing ability. No wonder then that the task of learning to speak comes so easily to the normally sense-endowed child. The language pathways are built up in the mind by the child's previous non-verbal thought, Piaget's "sensorimotor stage," that precedes speech, and not only does this ease the acquisition of spoken language, it obscures the mental activity underlying speech, pattern language.

> Language forms carry very little information per se, but can latch on to rich preexistent networks in the subjects' brains and trigger massive sequential and parallel activations. Those activated networks are of course themselves in the appropriate state by virtue of general organization due to cognition and culture, and local organization due to physical and mental context. Crucially, we have no awareness of this amazing chain of cognitive events that takes place as we talk and listen, except for the external manifestation of language (sounds, words, sentences) and the internal manifestation of meaning: with lightning speed, we experience meaning. This is very similar to perception, which is also instantaneous and immediate with no awareness of the extraordinarily complex intervening neural events.
>
> (Fauconnier 1999)

Indeed it has been said of natural languages that they *have no semantics* in terms of themselves, and that it is only the almost accidental existence of a 'key' containing coincidental resonances with a known language, such as the Rosetta Stone, that enables the unlocking of a language that has remained unused for many centuries.

> English inherits its semantics from the content of the beliefs, desires, intentions, and so forth that it's used to express, as per Grice and his followers. Or, if you prefer (as I think, on balance, I do), English *has no semantics*. Learning English isn't learning a theory about what its sentences mean, It's learning to associate its sentences with the corresponding thoughts. To know English is to know, for example, that the form of words 'there are cats' is standardly used to express the thought that there are cats; and that the form of words 'it's raining' is standardly used to express the thought that it's raining; ... and so on for in(de)finitely many other such cases. [Emphasis in original].
>
> (Fodor 1998, p. 9)

So the communication aspect of language comes after its use in understanding in the sequence of human development, and is, in fact, a *formalisation* of thinking. "I can only communicate with you to the degree that you and I make sense of the world in the same way" (Boyle 1971, p. 38). All human expressiveness, even rituals that seem impractical and useless to us, must, as Franz Boas has pointed out, have their origin in the need to "explain" the world (Boas 1963, pp. 198-9). There is no doubting the fact that these forms are functionally expressive, that is, linguistic in intent. Unfortunately, once the verbal capacity kicked in, it has tended to obscure the conceptual aspect of singing, dancing and other ritual forms, and in modern times we lean toward seeing only the verbal as conceptual in nature. Most communication is, to say the least of it, banal, and it's a pity that all the empty noise obscures the fundamental role of language in the ordering of our mental life.

> Education and society have greatly advanced the prestige of verbal skills. ...Let us call the characteristic verbalizing man, homo jabber. Now homo jabber has had everything his own way for a long time – it is not possible to read a newspaper without realizing his sovereign sway in human affairs, and a fine mess he has made of them. But it is at least a tenable argument that the highest forms of thought are not verbal at all, or, at least only insignificantly verbal.
>
> (French 1994)

And this sequence in the history of human development, thought before verbal communication, is reflected in the development of the individual. Long before we have begun to talk or write we have learned to *make sense* of the world, at least in a partial way, as it manifests immediately around us[1].

[1] "One particularly important quality of the human baby is its ability to learn. Endowed with its good range of sensory capacities - hearing, vision, taste, touch, smell, balance and

5.2 Making Sense of the World

This can be seen in much of the common behaviour exhibited by human infants. A newborn baby demonstrates behaviour that indicates a strong affinity for 'human face shape'. Very soon after birth infants are able to discriminate this 'form' in amongst what must otherwise be, to them, a pretty good imitation of visual chaos, especially given their limited ability to focus the lens of their eye. But even more than this, they associate this visual form that appears in their visual field with their own face, which has no existence in their own visual field. Savour that last sentence again. Here is a being, a newborn being, that is associating a form with the same form in its own anatomy, without any possibility of visual comparison - it is pure metaphor. Why does this not amaze us? This is a truly staggering ability when you think about the issue carefully. Yet the observation is commonplace. If you come into the field of vision of an infant and poke out your tongue or grimace, the baby will respond in kind, so, not only is it 'associating' similar form from the visual to an inherent component of itself that it can't see, it is motor co-ordinating the correct imitative behaviour without any training at all.

> Even new born literally new born, 42 minute old babies, can imitate the facial expression of another person. So if you stick your tongue out at the baby that baby will stick its tongue out at you and we could use that behaviour to find out something quite deep and intense about what the child knew. Namely that they already knew that there was a relationship between their own faces as they felt them inside and the picture they saw, the face they saw on another person.
>
> (Gopnik 2003)

The baby has constructed the following understanding where *that face* is the face shape that is present in its field of vision and *my face* is its own face which it cannot see - *my face* equals *that face*. This demonstrates that it has some sort of concept for face that is not embedded in visual form, or even, presumably, in symbolic form, and that it can put concepts together to form a larger understanding. It's just as if it had spoken the sentence, "that object that I see protruding from that face over there is the same as this (poking its own tongue out)". By imitating the behaviour that it sees it is clearly putting abstract concepts together in a very language-like manner *without the abstractive power of language.* This suggests that abstraction, like metaphor, is primarily a factor in cognition, not linguistic form. To be able to perform this feat at such an early age shows that it cannot have been simply 'taught' to appreciate a certain visual form. In other words babies show us that they understand far more than we have been used to giving them credit for.

temperature detection - the baby can start to monitor the world and learn how it is organised" (Morris 1991, p. 121).

The type of thinking, *conceptual thought*, that underlies this basic understanding is built on a language based, not on words, but on the mental concepts that we form around the objects and activities in our environment. It is not a coincidence that the word we use for our means of experiencing the world, *sense*, turns up in our concept for *understanding*. The phrase, 'understanding is making sense' of the world, underlines the recursive nature of the two concepts. Simple unfiltered sensory experience on its own is not enough for understanding, it is sensory experience shaped and guided by understanding that drives the human adventure - sensory experience given meaning.

> Unlike the filing cabinet, the brain is a self-patterning system. As soon as it comes across new information, it seeks ways in which it has come across such data before. It looks for similarities, parallels and metaphors. How active[ly] it seeks these is rather dependent on the skills and learning habits of the learner, but even the most unpracticed and slovenly thinker subconsciously has access to a vast and complex system of patterns, with immediate and elegant handling of the incoming information stream. The more patterns that have been, the easier it is to match yet more information. Moreover, the more diverse the types of patterns stored, and the greater the familiarity and frequency of access to them, the greater the learning potential. That is why adding a new pattern type is so critical to future learning; by doing so, we add not merely yet another fact, but a whole new way of seeing things. Future data could be transformed by such pattern. In this way, creating a new pattern is an investment in the future, even though at the time it cannot be predicted that the pattern will be ever used again.
>
> (Kaipa & Johnson 1999)

The process of learning to speak then builds on this fundamental level of language-as-conceptual-understanding. We conceptualise, understand the concept, before we verbalise, talk about it. To use a linguistic structure a child must understand the concept. As Piaget demonstrates with his set of dolls and sticks of various sizes a child cannot use comparison of size if the child does not understand the concept of size (Piaget & Szeminska 1952, p. 97). So it is not words that are the basis of language, but concepts; and concept is meaning.

> Meaning is an inalienable part of word as such, and thus it belongs in the realm of language as much as in the realm of thought. A word without meaning is an empty sound, no longer a part of human speech. ... Word meaning is both thought and speech.
>
> (Vygotskii 1962, p. 6)

The symbolic aspect that underlies language as a means-of-communication is secondary to semantics because the fundamental purpose of communication is to impart meaning. It is meaning, our understanding of a situation, that we are trying to communicate by the use of our language symbols and we *must have the meaning first*, before we can know what words or symbols to use. "Human language is much more than speech, it is a system for representing knowledge that lies at the very core of human thought (Beatty 1995). Language is fundamentally a system for expressing meaning internally within the individual's mental system on which the external spoken and written forms is built. The progression is from patterns of experience to generalisation and only then is communication possible.

> The world of our experience must be enormously simplified and generalized before it is possible to make a symbolic inventory of all our experiences of things and relations, and this inventory is imperative before we can convey ideas. The elements of language, the symbols that ticket off experience, must therefore be associated with whole groups, delimited classes, of experience rather than with the single experiences themselves. Only so is communication possible, for the single experience lodges in an individual consciousness and is, strictly speaking, incommunicable.
>
> (Sapir 1971)

We talk of language-as-communication as being a symbolic system but fail to consider the question 'what is it that a word symbolises?' This question can only be answered widely by 'meaning', as most words do not embody a direct relationship between the word and a particular object.

> A word does not refer to a single object, but to a group or to a class of objects. Each word is therefore already a generalization. Generalization is a verbal act of thought and reflects reality in quite another way than sensation and perception reflect it. Such a qualitative difference is implied in the proposition that there is a dialectical leap not only between total absence of consciousness (in inanimate matter) and sensation but also between sensation and thought. There is every reason to suppose that the qualitative distinction between sensation and thought is the presence in the latter of a generalized reflection of reality, which is also the essence of word meaning; and consequently that meaning is an act of thought in the full sense of the term.
>
> (Vygotskii 1962)

The relationship between word and meaning, when considered carefully, is much more diffuse than a direct one between word and object. In fact, we usually have to point, even if only verbally, when we want to talk about a particular object - *the* house, or *that* car - because the word itself is more about the concept, the meaning, than acting as a "sign" or "signifier of" a particular object or entity.

> Symbols are not proxy for their objects, but are vehicles for the conception of objects. To conceive a thing or a situation is not the same thing as to "react" toward it overtly, or to be aware of its presence. In talking about things we have conceptions of them, not the things

> themselves; and it is the conceptions, not the things, that symbols directly "mean." Behavior toward conceptions is what words normally evoke; *this is the typical process of thinking* [emphasis added]. Of course a word may be used as a sign, but that is not its primary role. Its signific character has to be indicated by some special modification - by a tone of voice, a gesture (such as pointing or staring), or the location of a placard bearing the word. In itself it is a symbol, associated with a conception, not directly with a public object or event. The fundamental difference between signs and symbols is this difference of association, and consequently of their use by the third party to the meaning function, the subject; signs announce their objects to him, whereas symbols lead him to conceive their objects. The fact that the same item - say, the little mouthy noise we call a "word" - may serve in either capacity, does not obliterate the cardinal distinction between the two functions it may assume.
>
> (Langer 1976, pp. 60-1)

So the symbols used in language-as-communication are almost totally about semantic purpose, the symbolic aspect is merely a means to the semantic end. We humans are generators of meaning, systems for understanding the world if you will, and in this regard we are completely different from computers, which are pure symbol manipulators. So, although on the surface it looks as though we do the same thing, process symbols, the results are startlingly different.

In the human case you end up with an understanding, whereas the computer's product is still just a symbolic representation of some kind. The result of the computer's prognostications can only jump the chasm to meaning through the agency of being understood by a human. The means might look similar in both cases but really humans are processing meanings, not symbols. This is totally unlike the computer's situation as the symbols carry no meaning except for the human one. It is a mistake, therefore, to call these things symbols, as in terms of the computer system itself they are just meaningless marks, as alien hieroglyphics are for us. The computer is manipulating marks that happen to be symbolic for our purposes, but completely meaningless to the computer itself. A computer is a tool to aid human understanding, and as such, it bears the same semantic relationship to the marks it uses as a piece of paper does to the marks that it carries - none. Moreover it can never develop a semantic connection because we humans start with meaning and develop symbolic form from that base, whereas computers start from symbolic form. The baby has no symbols, but it understands. Only later does it extend its means of understanding into symbolic form. There is no evidence that the reverse process, symbolic form to understanding, is possible, except by a living organism that has made the journey from understanding to symbolic language first. Indeed, it could be said that Helen Keller is one of the few exceptions that prove the general principle.

5.3 Patterns and Concepts

Syntax is an important aspect of any symbolic system, of course, but it is subsidiary to semantics. It allows us to generate strings of concepts, yes, but it is semantics that generates meaning larger than that that can be expressed in single concepts. Here again is that fundamental connection with patterns, the grounding of ideas, which exist in the mind, with patterns of real world experience. A generated string of concepts will only make sense if it fits a pattern of experience. But what do we mean by this? It turns out that making sense of something is almost entirely contextual. What we are really saying when we state that something 'doesn't make sense' is that it 'doesn't make sense in the context of our everyday experience of the real world.' For example, the syntactic rules of our spoken language allow us to generate the sentence "the cat sat on the mat" and our semantic checker will pass this because it makes sense as a string of concepts, it has meaning in terms of the real world. However the sentence "the mat sat on the cat", although syntactically correct, would normally be rejected as a string of concepts as it has no meaning based on real world experience.

But this rejection is entirely contextual. What we are basing the rejection on is the fact that in the context of everyday experience, mats do *not* sit on cats, there simply isn't a real world pattern for this. Given a different context, however, a fantasy or dream-like experience such as "Alice in Wonderland", for example, where rabbits speak English, a pattern like this would possibly be acceptable. In other words, patterns are all context. Calling something a pattern is just saying that given a certain set of conditions, a context, this experience is *likely.* And anything that is likely under certain circumstances is bound to recur when the circumstances are the same. It's hard to imagine a better definition than this of the pattern idea! So in the sense that understanding is a type of meaning, it too is contextual through the concept-pattern nexus.

Even spoken language, then, is based on concepts not words as such. It is the meaning of a word, the concept it symbolises, that is the important aspect. But concepts are not themselves part of the external reality that we experience, they are mental artefacts that we construct in order to build up the mental model of reality that is our means of understanding the world. The question then is, 'from whence cometh the concept?' and the answer is from repetition in our experience of the world. Repeating form is pattern, and so concepts derive from patterns in reality. "Without such patterns the world appears to be such an undifferentiated homogeneity that man is unable to make any sense out of it." (George Kelly, 1963 quoted in (Britton) 1970, p. 17) Language-as-conceptual-understanding, then, is essentially a pattern language. "When we were born, the world, fresh, contained no repetitions. An infant first smiles at his mother's return not because it is her but because her presence repeats; later he smiles because the she who returns is the same she" (Gabriel 1996*a*, p. ix).

It is the simple repetition that gives rise, over time, to the concept of 'she.'

Without the pattern there would be no possibility of the abstract concept.

> Recurrence works because actions, characteristics, and relations that are noticeable repeat. Every act - every thing - contains not only the noticed parts but also attendant, variable portions. The recurrent parts stand out because our minds (our brains?) are constructed that way ... Repetition is power. What is important repeats and reveals commonality. What isn't repeated is variable and can be highlighted or hidden. ... Abstraction reveals and revels in what is common, the repetition, and what is abstract is defined by the repeated.
>
> (Gabriel 1996*a*, p. ix-x)

What we are doing, then, from the moment we are born, is understanding the world[2]. This is the continuous process by which we make sense of our changing circumstances. The way we put this is English is instructive. Both understanding and making sense are primarily verbs, which indicates that we are talking about a process. Living can be seen as a lifelong problem solving process, the problem being to make sense of our current situation - "learning is a lifelong enterprise"(D'Arcangelo 2000). As animals the primary driving force is survival, but as humans we have extended our capabilities for understanding things far beyond that required for simple survival and this extension is the basis of the human condition. All animals have to make sense of the world to some extent. We humans have made it our defining characteristic, it is essentially this feature that distinguishes us from other species. Understanding is not merely a means of survival, it is our work, our pleasure, our raison d'être, the very stuff of our being. The 'need' to understand, to build a conceptual image of the world in which we live is the motif of human life, and concept is the basis of the recognition of what is the same and what is different, that is differentiation, the precondition for any ability to relate things, and therefore the basis of any combinatorial process be it understanding, communication or manual creativity.

> Concepts are the building blocks of thought. Without them, induction would be impossible because everything would be unique. Different things have to be treated as the same for some purposes, and similar things have to be treated as different for other purposes. Concepts provide the system for classifying, subdividing, and interrelating things.
>
> (Johnson-Laird 1993, p. 87)

The processes that build on this basic "system for classifying, subdividing, and interrelating things" (*meaning system*) - cognitive, affective and emotional -

[2] "Cognitive developmental psychologists are now actually looking into the crib to study the development of the mind early on, even before children develop language. And what we find there is very interesting. We find a little scientist peering back at us - a child who is desperately interested in making sense of the people, the objects, and the language around him or her, a child doing mini-experiments to try to sort everything out" (D'Arcangelo 2000).

are really just expressions of the power of concept formation. This fundamental correspondence between the various features of the human mind, is obscured over time as they *emerge* as systems in their own right.

> In studying man's earliest history, when the evocative qualities of certain forms and the power of symbolism in non-random shapes and sounds was being discovered, it is difficult to separate things done for "pure" aesthetic enjoyment from those done for some real or imagined "practical" purpose. The man who selected for admiration a beautifully shaped and textured stone was yielding to a purely aesthetic motivation, but the man who molded clay into a fertility figurine was simultaneously an artist, a scientist learning to understand the properties of matter, and a technologist using these properties to achieve a definite purpose. ... [The fact is] that in the earlier stage of discovery, first of form and later of materials that, once shaped, would retain desirable form, the motive can hardly have been other than simply curiosity, a desire to discover some of the properties of matter for the purpose of internal satisfaction. Paradoxically man's capacity for aesthetic enjoyment may have been his most practical characteristic, for it is at the root of his discovery of the world about him, and it makes him want to live. It may even have made man himself, for, to elaborate a remark by the poet Nabokov, it seems likely that verbal language (to which anthropologists now assign vast evolutionary advantage) was simply a refined use of the form-appreciating capabilities first made manifest in singing and dancing.
>
> (Smith 1981, p. 194)

If understanding really is based on this idea of language in the widest sense then it seems necessary to explain this. The connecting point, we believe, is meaning. Essentially a concept is a concept because it encapsulates meaning. Whatever it is that is being conceptualised has significance in terms of our existence. Without such significance there would be no point in expending mental energy on it. If a concept was just pure thought, if it had none of that correspondence to something in reality that we associate with the notion of concept, then it would be meaningless. Meaning is the essential connection between the mental world in our heads and the real world, it is meaning that makes a particular thought a concept - "the missing relationship between thought and reality is 'meaning" (Vygotskii 1962, p. 10).

The power of concept derives from its two-sided nature. A concept arises from a pattern in a particular context but because it exists only in the mind - that is, it is fundamentally abstract - it can move into other domains where this is useful in terms of building understanding.

> The power of understanding symbols, i.e. of regarding everything about a sense-datum as irrelevant except a certain form that it embodies, is the most characteristic mental trait of mankind. It issues

> in an unconscious, spontaneous process of abstraction, which goes on all the time in the human mind: a process of recognizing the concept in any configuration given to experience, and forming a conception accordingly. That is the real sense of Aristotle's definition of man as "the rational animal."*Abstractive seeing* is the foundation of our rationality, and is its definite guarantee long before the dawn of any conscious generalization or syllogism. It is the function which no other animal shares. Beasts do not read symbols; that is why they do not see pictures. We are sometimes told that dogs do not react even to the best portraits because they live more by smell than by sight; but the behavior of a dog who spies a motionless real cat through the window glass belies this explanation. Dogs scorn our paintings because they see colored canvases, not pictures. A representation of a cat does not make them conceive one.
>
> (Langer 1976, p. 72)

So a concept is an idea that captures information about a situation that is partly, at least, independent of context. For example the term 'underdog' is context-free in the sense that it is routinely applied to situations in which no dogs exist, such as the David and Goliath story. Indeed, the very 'power' of this story is that the obvious "underdog", David, prevails. So the concept applies in any situation involving actors of any kind where a success-or-failure result is expected to arise from the situation. It is a concept involving the relative apparent status, in terms of the competition activity, of two competing entities. It requires a situation involving competition and an obvious discrepancy in ability in the competition activity, but it does not require the discrepancy to be real, only apparent, and it specifies nothing about the type of entities or competition involved. In other words it is a concept precisely because of its general applicability to many different situations, because it is, to some extent, context free. And this makes it widely useful. It can be used to partially describe a complex situation - an apparently one-sided competitive one, in this case - in a concise and precise manner.

Thus a concept is more about relating things than describing them directly - the understanding of a situation arises through the analogy with another, not a direct description. With the 'underdog' concept we understand that here is a situation that is analogous to a fight between two dogs of grossly uneven size, even though the words 'fight', 'dogs', 'size' and 'under' are irrelevant in a literal sense. So the context for the use of a conceptual term is general, except in terms of the factors that make it a concept, the meaningful ones. It is specifying those attributes that are contextually important in the system being conceptualised, that have 'meaning' in terms of the system. For example, there is no inherent reason that the term 'underdog' could not stand for 'one dog being physically underneath another one', but we know that it doesn't carry that meaning in terms of the English language seen as a communication system. Its power derives from what it means in a system-contextual, not a literal, sense. Concepts help us "interpret our experience by organising it in terms of distinctions"(Smith 2003).

Conceptual meaning, then, is not necessarily literal, but system-related. The difficulty with concepts is that we are using words, elements of a communication system, to symbolise, that is, stand for, them. So the concept of an 'underdog' is being symbolised by the combination of two words, neither of which, in combination, is being taken literally. It embodies a meaning relationship in terms other that the language system itself. The meaning of the combination is coming from somewhere other than the literal meaning of the constituent parts, and that somewhere is the analogy with the uneven-appearing dogfight situation. Thus the important aspect of the conceptualising process is the relationship of the part to the whole. It is the power of abstraction - the making of the decision about what is an important aspect of a situation in a particular context. The concept tells me what aspects of an entity are meaningful in terms of the system, and therefore what relationship it holds to other aspects of the system.

In particular, this essential relationship between a particular concept and the "system" of which it is an element is illustrated by the different response to similar "natural" phenomena exhibited by different cultures. People who live in "advanced" societies tend to see such phenomena according to the "scientific" framework that provides the "atmosphere" of modern thinking - the existence of "matter and force". But this is not the case with societies in a prescientific relationship with their environment.

> It is vain to try to understand primitive science without an intelligent knowledge of primitive mythology. "Mythology," "theology" and "philosophy" are different terms for the same influences which shape the current of human thought, and which determine the character of the attempts of man to explain the phenomena of nature. To primitive man - who has been taught to consider the heavenly orbs as animate beings; who sees in every animal a being more powerful than man; to whom the mountains, trees and stones are endowed with life or with special virtues - explanations of phenomena will suggest themselves entirely different from those to which we are accustomed, since we still base our conclusions upon the existence of matter and force as bringing about the observed results.
>
> (Boas 1963, p. 200)

Concepts, then, are not simply elements of understanding that stand isolated from the context in which they arise. Rather they are part of a complex web of 'meaning' relationships.

> From an analysis of a significant fragment of the English lexicon, George Miller and I concluded that lexical meanings are organized into semantic fields. Underlying such a semantic field is a conceptual core, a theory-like structure that integrates the different concepts in

> the semantic field around one or two core concepts, such as color, kinship, motion, possession, and perception. These concepts are interrelated by concepts that occur in many fields, including concepts of space, time, possibility, and permissibility. The theory of a conceptual core underlies the paramount fact that concepts are not isolated mental tokens. They are organized taxonomically by the conceptual relations into which they enter. These taxonomies enable us to categorize things without having to be equipped with precise concepts. Analytic concepts are precise, but natural kinds and constructed entities are not, so the boundaries between their instances cannot be mapped with absolute precision.
>
> (Johnson-Laird 1993)

This "original" or "primary" contextual aspect, the "proto-meaning" in effect, is the important factor in the building of subsequent understanding, and it is the ability of concepts to float across domains that underlies the notion of metaphor. But notice that it is the 'meaning' that is mobile rather than the symbol, the meaning, the "contextual understanding" if you will, carries the symbol along with it. So, for example, the "underdog" concept that we discussed earlier carries the meaning "appears likely to lose" from the dog fight context, to one in which neither the word 'dog', 'fight' nor 'under' are literally appropriate. It is the *meaning* that is appropriate, the *pattern.* Metaphor is thus not, primarily, a linguistic phenomenon but a *cognitive process.*

> Metaphor [is] a pervasive mode of understanding by which we project patterns from one domain of experience in order to structure another domain of a different kind. So conceived metaphor is not merely a linguistic mode of expression; rather, it is one of the chief cognitive structures by which we are able to have coherent, ordered experiences that we can reason about and make sense of. Through metaphor, we make use of patterns that obtain in our physical experience to organise our more abstract understanding. Understanding via metaphorical projection from the concrete to the abstract makes use of physical experience.
>
> (Johnson 1987, pp. xiv-xv)

Metaphor, because of its abstractive power, can thus be seen as underlying "much of out complex thinking, reasoning, and language" (Smith 2003), that is, in fact, our mental world. Moreover, advances in the study of neurology have meant that it is now possible to make at least some connections between neural and mental activity. Work in this field backs up the notion of language being fundamentally about understanding rather than communication. For example, it now appears to be very likely that processes like abstraction, metaphor and creativity that were previously regarded as features of language can be related to particular components of the brain (the angular gyrus in this case).

> The various senses report their information to a central area for correlation and the creation of a central image of the particular object in question. The location of this "neo-cortical crossroads" was suggested by Harvard's Geschwind: "In man, the cortical association areas of the brain, each responsible for interpreting the senses of vision, touch and hearing, are clustered round and plugged into a structure known as the angular gyrus"(Desmond, p. 111). ... The angular gyrus ... makes a mess out of nominalism - because the word is no longer necessary as the origin and support of the universal. It appears to support Furth's remark that the "internal organization of intelligence is not dependent on language."
>
> (Plank 1998)

This reinforces the idea that abstraction, metaphor and creativity are not emergent properties of speech, but rather these "cross-modal" associations are the precursors of speech (Geschwind 1974). Associating and combining ideas, then, is the key to apprehending the source of human intelligence. Before one can go beyond the contemplation of concrete objects one must have abstracted their essence in mental form from the concrete reality of which they are a part. This is more than classification and the simple operations, such as counting, which go along with it, this is the very basis of reasoning.

> The exercise of logical reason is always concerned with these absolutely general conditions. In its broadest sense, the discovery of mathematics is the discovery that the totality of these general abstract conditions, which are concurrently applicable to the relationships among the entities of any one concrete occasion, are themselves interconnected in the manner of a pattern with a key to fit. This pattern of relationships among general abstract conditions is imposed alike on external reality, and on our abstract representations to it, by the general necessity that every thing must be just its own individual self, with its own individual way of differing from everything else. This is nothing else than the necessity of abstract logic, which is the presupposition involved in the very fact of interrelated existence as disclosed in each immediate occasion of experience.
> The key to the patterns means this fact:- that from a select set of those general conditions, exemplified in any one and same occasion, a pattern involving an infinite variety of other such conditions, also exemplified in the same occasion, can be developed by the pure exercise of abstract logic. Any such select set is called the set of postulates, or premises, from which the reasoning proceeds. The reasoning is nothing less than the exhibition of the whole pattern of general conditions involved in the pattern derived from the selected postulates.
>
> (Whitehead 1964*b*, pp. 13-4)

This notion of reasoning as the "exhibition of the whole pattern of general conditions involved in the pattern derived from the selected postulates" illustrates that most of creativity in thinking is an extension of what is already known. When Einstein made his famous breakthrough to understanding relativity, it occurred not in developing any new information but in taking the existing configuration of seemingly contradictory information and making sense out of it. What was new was the particular combination of the various pieces not the basic information itself. Einstein once explained how he saw his achievement in a letter to Jacques Hadamard, a famous contemporary mathematician, in these terms.

> The psychical entities which seem to serve as elements in thought are certain signs and more or less clear images which can be "voluntarily" reproduced and combined. ... But taken from the psychological viewpoint this *combinatory* [emphasis added] play seems to be the essential feature in productive thought - before there is any connection with logical construction in words or other kinds of signs which can be communicated to others. The above mentioned elements are, in my case, of visual, and some of muscular type.
>
> (quoted in (French) 1988 p. 283)

It is clear that the activity being described, "this *combinatory* [emphasis added] play", is that of language but the elements that are involved in the activity are, equally clearly, not of a type, that is, words, "which can be communicated to others." This is language-as-conceptual-understanding that Einstein is describing.

5.4 Why is Experience Rational, Ordered and Repetitive?

If knowledge consisted only of that which we experience through our senses then we would have to explain why our senses are continually recording the same sensory events. Given no underlying substrate to experience it should be the case that anything could happen. But our experience tells us that this is not the case, that there are well defined sensory events that keep occurring over and over again with differences in detail but otherwise essentially the same. Some factor must underlie this repetitiveness in sensory experience and the simplest explanation is that experience is located in an objective world rather than a purely subjective one. The fact is there are many possibilities, pigs flying, for example, that *never* occur in my experiential system. If my experience of pigs is purely subjective then there is no reason, in principle, why I could not experience them flying; all of life, not just the dreaming component, would be as a dream where nothing is inherently impossible. So the subjectivist argument must come up with an explanation for the fundamental limitedness of sensory experience in order to be taken seriously.

Moreover we could not talk, or even think, about our sensory experience given the absence of such repetition. A concept, and therefore a word, is nothing but a mental representation of a common factor of sensory experience, and if experience was purely arbitrary rather than ordered and repetitive then we could have no concepts. If I only ever had the experience of seeing a tree once then I would be very unlikely to retain any notion of having had the experience, let alone have a concept of 'tree' in my consciousness. This is because even our memory is based on concepts and words, and again there is a parallel in dream states. Often we have dreams that are very clear in their 'logical' temporal flow while we are dreaming but which fall apart when we try to remember the flow after waking, they no longer make the sense that they seemed to make while they were happening[3] (Gallagher 2000). This suggests that there are things that happen in dreams that fit no pattern, there is no previous experience in our memory that we can use to associate them with. Essentially they have no meaning. We cannot fit them into our 'understanding' in the widest sense, so the 'understanding' of the dream that we had while it was occurring fades once we are awake. Without the 'missing' bits the temporal flow is impossible to reconstitute and it all gets jumbled up in our memory.

Patterns, then, are the building blocks of that continuum that we know as intelligence. They lie at the very foundation of classification, discrimination and abstraction, the bases of the ability to conceptualise.

> All higher organisms display an ability to deal with variability in sensory stimulation. That is, they can assign nonidentical stimuli to classes, the members of which are more or less equivalent in terms of their implications for the organism's behavior. This ability is basic to human thought and communication. Indeed, it is doubtful whether anything worthy of the name of either thought or communication could exist without conceptual categories. Suppose that there were no such concept as "chair," and that each one of those patterns of sensory experience that we now classify as instances of "chair" had to he treated as a unique thing. Whenever one had occasion to refer to such a thing, one would have to describe it in terms of its features. But now we find ourselves thinking in terms of backs, and seats, and legs, which is to say in terms of other conceptual categories. If we disallow these, we push ourselves back to other categories that are perhaps somewhat closer to the sensory data - contours, colors, sizes, textures - but are conceptual categories still. To divest ourselves of the use of categories is to divest ourselves of thought.
>
> (Nickerson et al 1985, p. 9)

So despite the fact that experience is personal and subjective it is conditioned by the patterns of experience, and these patterns define the objective world of reality. Every person has a unique view of the world that derives from their individual experience of it and this factor is made quite complex by the fact

[3]This is why it is important in any psychological research on dreaming states for the subject to immediately write down their impressions upon awakening.

that there is a feedback loop involved - experience of the world is affected in many ways by the particular view of the world held by the perceiver. Having an interest in a particular field, for example, means that one is likely to actively colour one's own experience with that interest. But the subjective viewpoint is, at its fundamental level, based on two simple concepts, objects and events. The world is just 'objective reality', a collection of real objects that exist. By means of simply being in existence, objects will have relationships between them. For example the location of an object is defined, most commonly, by its 'spatial' relationship to other objects. So the very fact that there are a number of objects creates the concept of 'space'. Without any objects with location relationships between them, there would be no need to invent the idea of space. But once you have the idea of space you have the basis for a process, movement. The movement of objects can then be seen as an expression of change in this spatial relativity to each other. But considering movement has added another factor to our description of the objective world, and that is time, or rather the flow of time. So time and space are relations, "it is impossible to express spatio-temporal truths without having recourse to relations involving relata other than bits of matter" (Whitehead 1964*a*, p. 21).

At its base level, existence in space requires no notion of time. If some object, and the nucleus of a hydrogen atom is a prime example, simply exists, that is, partakes in no activity, then, it is, in principle, timeless. It is activity that creates the factor that we know as time - as Whitehead says, "time is known to me as an abstraction from the passage of events" (Whitehead 1964*a*, p. 34). Without a process, time is not dimensional. It is process that makes time a dimension, that adds quantity, gives it its measurability. And the most fundamental of all processes is movement - a change in the location of an object relative to other objects. For one object to alter its spatial relationship to other objects requires something additional to the three basic measurable spatial dimensions. You can never define the movement of an object in terms of the three spatial quantities alone, you need a fourth, and this is time[4]. So, in a sense, time is basically just change in location in space. It can be seen as spatial in the sense that it is spatial difference, and as extra-spatial, in that it is change in spatial difference over some other measure, time itself. So time only makes sense as a measure of difference in state, it is not time itself that flows, it is the state of the "material" of the system that changes. "Time is always the present. There is no other" (Harth 1993, p. 8).

This was the real import of Einstein's notion of relativity. Space and time are defined by the matter contained in the universe, and this is quite different from the classical Newtonian notion of 'empty space' as a sort of 'receptacle' in which matter 'occurs'[5].

[4]In a sense, time, like temperature, "is an emergent quality of nature defined only for large enough systems ... ... having no real existence in the quantum world of individual particles and systems" (Odenwald 2004).

[5]It is interesting to note that the Einsteinian conception harks back to the pre-Newtonian

> Einstein took some trouble to explain why his kind of space is so different from Euclid's and Newton's. He said that you could imagine a universe in which space was not warped, but it would be completely devoid of matter. Empty space has no practical meaning: space cannot exist separately from 'what fills space' and the geometry of space is determined by the matter it contains. For some readers, the explanations of the master should be sufficient; for others, I shall attempt a modern elaboration. The way in which time, space and the speed of light are linked together is not mysterious in principle, even though its consequences are weird. Any speed is defined as a certain distance travelled through space in a certain time. But in Einstein's universe the speed of light is more fundamental than space or time. Space is what light moves in; time is how long it takes to move.
>
> (Calder 1979, pp. 92-3)

As concepts, then, space and time are not in the least abstract. As soon as you have material existence, that is objective reality, or saying it another way, real objects, you have space and time, which are just the measurable relationships between objects. But the fourth dimension necessarily involves change, it derives from a process, an event. If there were no events there would be no time. Existence now involves more than just simply existing in space, the existence of multiple objects implies relationships between them, and spatial relationships create the potential for change, for movement in space. Or as Einstein said, "spacetime does not claim existence on its own but only as a structural quality of the [gravitational] field" (Einstein 1962, p. 155). Therefore objective reality is just the real world as it is defined by the existence of 'material' things. The concepts that arise in thinking about it are tied, pretty tightly, to objects and relationships, and are hardly at all abstract in the sense of being separate from material instance. There could be no concepts of space and time without the 'material' existence of multiple objects to be perceived by our mental system. "Time and space are modes by which we think and not conditions in which we live" (Einstein quoted in (Forsee) 1963).

Field theory, or "local causality," is the clearest principle that arises from Einstein's prognostications. It addresses the great metaphysical issues that arise in any contemplation of the cosmos, 'time' and 'force.'

> The world of objects may be said to lack both past and future. Only the present exists, but it is an existence of total isolation in space and

one that derives from the common sense experience of the world."The most general notions underlying the words 'space' and 'time' are those ... aimed at expressing ... their true connection with the actual world. The alternative doctrine, which is the Newtonian cosmology, emphasized the 'receptacle' theory of space-time, and minimized the factor of potentiality. Thus bits of space and time were conceived as being as actual as anything else, and as being 'occupied' by other actualities which were the bits of matter. This is the Newtonian 'absolute' theory of space-time, which philosophers have never accepted, though at times some have acquiesced" (Whitehead 1929, p. 97).

> time for every speck of dust, for every atom in the universe. The conditions of this moment cease to exist a moment later, and no future event exists before it becomes the present. No influence is exerted across finite time gaps, just as, in space, there is no action-at-a-distance. Present events result from present conditions, that is, conditions that are contiguous in space and time. The past has dropped out of existence. It is only the equations of physics, or our less rigid memories and intuitions, that allow us to extrapolate over an infinity of such infinitesimal contiguities and to make statements about past or future events.
>
> (Harth 1993, pp. 8-9)

It might be pointed out here that there is a flaw in the non-abstract nature of objects because even the concept of 'material' itself is 'fuzzy'. As David Bohm points out, the implication of quantum phenomena is that matter at the subatomic level is not matter-like (particulate) in the normal classical sense, but more like a process (a wave) taking place in a general medium (a field) (Bohm 1962, p. 65). However this realisation only impacts on the representation or understanding of reality. Pragmatically the situation is, as Niels Bohr pointed out through his 'correspondence principle', exactly as it was before we discovered quantum indeterminacy. The assumption is that there exists in nature a wave function, or state vector, that represents, or corresponds to, the matter-like aspect of reality - that is, not just a mental representation but a 'real', independent of mind, effect. So these things are not abstract in the real meaning of that word, it's just that "instead of beginning with space and time, as if they were fully in existence, and then placing various objects in it, we first begin with the whole process as it is and then try to discuss the order of things in space [and time] ... since I am discussing the topology of process."(Bohm 1962, p. 77) In other words, objects, space and time are all aspects of a whole system, 'objective reality' - these things can't be separated from each other in any real sense. We mistake the process of breaking things into smaller parts as an aid to understanding as a reflection of actual fragmentation rather than a product of mental representation (Bohm 1980, pp. 2-5).

This is not meant to imply that the decomposition process has progressed on the basis of purely arbitrary divisions, that there is no basis in reality for the method we employ in understanding the world. Probably the central notion of pattern language is that form is the product of context. "If the world were totally regular and homogeneous, there would be no forces, and no forms. Everything would be amorphous. But an irregular world tries to compensate for its own irregularities by fitting itself to them, and thereby takes on form" (Alexander 1964, pp. 15). Ultimately it is different form, the expression of forces in different contexts that sets up further irregularities and therefore more complex contexts in which forces are again resolved by form, setting up further irregularities that result in new forces in new contexts, and so on, ad infinitum. So, it would seem to be a property of reality to create complexity, and that therefore "there are special sciences not because of our epistemic relation to the world, but because of the way that the world is put together" (Fodor 1975, p. 24). This is the force of Bohm's "topology of process". If it is anything at all, the universe is a process that creates time and space, it *designs* reality in Alexander's sense of deriving form from the resolution of forces in a context.

5.5 Subjectivity - A View of the Objective

So, in terms of the objective base of reality, then, what is this 'subjective' thing? In essence it is nothing more than awareness of objective reality, a view of objects and the relationships between them. Just as it is the relationships between objects in space that creates the potential for change, and therefore the concept of time, so the relationship between time and space creates the potential for awareness. This is because it is not just spatially that the relationships between objects can now change. Given that time exists as a dimension, factors other than spatial ones can also change. At the atomic level the potential to change location brings with it possibilities like aggregation, which allows things like the exchange of electrons between atoms to occur, and for gravity to become a factor in super-atomic as well as subatomic space. And once the relationship between objects is more than just spatial, *the basis for evolutionary change is laid.* The dynamics of a system of objects is now chemical as well as physical and spatial. In turn, this chemical base provides other potentials for change through process, and the type of events that can occur in a system. So material existence is the basis of space, space is the basis of time, time is the basis of chemistry, chemistry is the basis of biology, and biology is the basis of awareness.

Thus a basically simple system, consisting of a set of material objects, is the source of all the complexity that we see around us. But more than that, it is the very basis of the process of awareness of the complexity around us. In a vital sense, awareness, and therefore the subjective experience of the world, is merely an extension of the relationships between objects. In order to explore this idea, consider a simple world that consists of multiple objects and one type of relationship between objects, based on the occurrence of an event called, in the example, event type 1, 'e1'. So a relationship between two objects consists of occurrences, single or multiple, of this one kind of event. Table 5.1 lists all the events that have occurred in this system up to the current time, by linking the two objects involved in each event. Objects are represented by letters and linked by a dash symbol. The flow of time from the beginning of the system to the present is represented by the sequence of table elements a1 to c5.

In the sequence given there are only two objects that have relationships that consist of more than one event occurring between them, A and B, so we will consider just these two relationships. The relationship list for object A consists of

	a	b	c
1	A - B	A - F	A - I
2	A - C	A - G	A - B
3	A - D	A - B	A - J
4	A - B	A - B	A - K
5	A - E	A - H	A - B

Table 5.1. Sequence of events between objects through time from a1 to c5.

the entire list, whereas that for B consists only of the events involving both A and B. So these two relationships form the basis for two different views of the system. From the table it can be seen that B has a multiple event relationship with A, and A has a multiple event relationship with B, and single event relationships with C, D, E, F, G, H, I, J and K. If awareness in this system was to be built up from multiple events for an object, then it would be based on these two relationships, as all the other objects have only single event relationships, there are no repetitions, no patterns of behaviour on which to build awareness.

In other words, there are only two possible views of this simple world, the view of A, which is based on multiple occurrences of event e1 with several other objects, and the view of B, which consists of multiple e1 events with object A. So A could be said to be involved in an object-promiscuous relationship of type e1 events, and B in a single-object e1 event relationship, and any 'awareness' would reflect these - the world, for B would consist only of A, whereas A's world contains multiple objects. The same 'objective reality' can be seen, then, to provide scope for multiple subjective views simply on the basis of the existence of different objects.

Think about a family relationship. The view of a particular family for an outsider will be quite different to the view of it held by the various family members themselves. The outsider does not partake in any of the relationships that constitute the family ties, and can therefore be 'objective' about the family in a way that insiders can't. The relationships are almost entirely subjective for the insiders. Moreover, to the insiders, the family circumstances will appear as normal because it is the family that they know best, and so it provides the main input to their measure of normalcy. However this is not true of the outsider, who can therefore form a quite different view that is not affected by involvement in any of the relationships that make up the family. The outsider view is coloured by the differences between the family and her view of family normalcy, in a way that the insider view cannot. And this effect is magnified at larger levels of group relationships, forming the basis for cultural suspicion and misunderstanding between communities.

At base level then, subjective awareness within a system, is still just the fundamental differentiation of objects. All that has happened, really, is that objects that have awareness, subjects, see the world through their relationships with other objects. Nothing that is fundamental to the system of objects has changed. Before A and B acquired 'awareness', there was only one possible 'view' of this world, the god's eye view, so to speak, given in Table 5.1. But this is just the objective reality itself, a map of the material existence of the objects and the interactions between them, nothing more. So the subjectification of the system involves the arising of viewpoints internally to the system - the 'subjective' is just an object's view of the system, rather than a godlike view. It is only different from objective reality by means of viewpoint. So we are back to the spatial relationship again, a 'subjective view' can be primarily based on nothing more than differentiation in space. We still haven't moved from the spatial implications of a system of existing material objects, we have just started looking from within the object space rather than from without. Big deal!

What is happening is that at some point in the development of life, awareness arises, information and meaning are created so to speak, and this involves the primeval 'epistemic cut', the distinction between subject and object.

> The cut itself is an epistemic necessity, not an ontological condition. That is, we must make a sharp cut, a disjunction, just in order to speak of knowledge as being "about" something or "standing for" whatever it refers to. What is going on ontologically at the cut (or what we see if we choose to look at the most detailed physics) is a very complex process. The apparent arbitrariness of the placement of the epistemic cut arises in part because the process cannot be completely or unambiguously described by the objective dynamical laws, since in order to perform a measurement the subject must have control of the construction of the measuring device. Only the subject side of the cut can measure or control.
>
> (Pattee 2001*b*)

As von Neumann demonstrated the distinction between subject and object *requires* a functional description of the execution of control and measurement processes that is, in principle, *not* reducible to the elements of the system being controlled or measured. In simple terms, life entails a disjoint that is not distinguishable at the level of physical law, it requires the storage and transmission of 'information', a semiotic system - that is, a heritable generic memory is a prerequisite of life, "information must be acquired and *used* for survival. Otherwise it is entirely gratuitous to attribute function, fitness and meaning to biological structure" [emphasis in original] (Pattee 1996).

> Physical laws and semiotic controls require disjoint, complementary modes of conceptualization and description. Laws are global and inexorable. Controls are local and conditional. Life originated with semiotic controls. Semiotic controls require measurement, memory, and selection, none of which are functionally describable by physical laws that, unlike semiotic systems, are based on energy, time,

> and rates of change. However, they are structurally describable in the language of physics in terms of nonintegrable constraints, energy degenerate states, temporal incoherence, and irreversible dissipative events.
>
> A fundamental issue in physics, biology, and cognitive science is where to draw the necessary epistemic cut between the coherent physical dynamics and its rate-independent semiotic description. To function efficiently, semiotic controls at all levels must provide simple descriptions of the complex dynamical behavior of the input/output systems we call sensors, feature detectors, pattern recognizers, measuring devices, transducers, constructors, and actuators.
>
> For sufficiently complex temporally incoherent semiotic control behaviors (e.g., designing, forecasting, choosing strategies, policy making, coevolution) there is at present no known procedure for finding those input patterns whose recognition provides the most efficient control for long-range survival other than by an evolutionary process of search and selection.
>
> (Pattee 1996)

So the 'epistemic cut' is significant in terms of the notion of 'objective knowledge' as well. Although in some sense all knowledge is subjective, the description/construction or genotype/phenotype relation must exist as physical reality if biological evolution is to occur - a semiotic description of physical construction is logically necessary for systems that can indefinitely increase functional complexity, that is, that display evolutionary processes. There is some sense, then, in which semiotic form precedes subjective awareness and it is the same relation between information and reality exposed in biology that provides Eddington's "hard core of objectivity behind the colorful tale of the subjective storyteller mind" (quoted in (Pattee) 2001). Making sense might be a subjective process, but what is being made sense of is objectively real, or there is no way that sense can be made.

But there is an important side issue here as well. The basic physical nature of the transmission of 'genetic' information through protein molecules forms the foundation of the holistic nature of a living organism. "A single folded protein has no function unless it is a component of a larger unit that maintains its individuality by means of a genetic memory. We speak of the genes controlling protein synthesis, but to accomplish this they must rely on previously synthesized and organized enzymes and RNAs" (Pattee 2001*b*). The concept of 'information' similarly implies a holistic relation between mind and the environment in which the organism that hosts it exists - information is grounded in reality or it is essentially meaningless, that is, it is *not* information.

This is why the subjective - objective split is epistemic rather than ontological or even metaphysical. The point is that there are different levels of informational awareness being expressed by living form and even occurring simultaneously in the human brain, but the only level that we can talk about is the level that we call 'consciousness'. This level is the subjective level, the 'I know' sense of knowledge, that belongs to what Popper calls the 'second world', the world of subjects, as opposed to the 'first world', the world of objects.

> [Thus Poppers'] thesis involves the existence of two different senses of knowledge or of thought: (1) knowledge or thought in the subjective sense, consisting of a state of mind or of consciousness or a disposition to behave or to react, and (2) knowledge or thought in an objective sense, consisting of problems, theories, and arguments as such. Knowledge in this objective sense is totally independent of anybody's claim to know; it is also independent of anybody's belief, or disposition to assent; or to assert, or to act. Knowledge in the objective sense is knowledge without a knower: it is knowledge without a knowing subject.
>
> (Popper 1972, pp. 108-9)

Popper's realm of 'objective knowledge', which he says is 'totally independent of anybody's claim to know' is therefore analogous to Plato's world of ideas, at least in that respect.

Subjective knowledge is a dynamic relationship that exists between a mind and the world around it, and is in a continual state of development. It lives or dies with the organism in which the mind resides. Objective knowledge is quite different. It survives the death of a single organism precisely because it is not tied to a single subject; it is not subjective. Once knowledge of natural numbers exists, the existence of odd and even numbers follows. But there is nothing subjective here, even though a number system is completely abstract in itself. The various characteristics that numbers display, like oddness and evenness, are properties of sequence, an inherent characteristic of order, not something that is tied to any particular mind, or state of mind, or even number theory. An innumerate shepherd might imagine that he has put as many of his flock in one field, as another, by directing them, one through the left gate, and the next through the right, and so on, as they pass by him, but if his flock consists of an odd number of sheep then it is simply an objective fact that he cannot do this, and his subjective lack of knowledge about numbers makes not a whit of difference. Numbers follow sequence, not the other way around. The subjective understanding arises out of the objective reality

Alfred North Whitehead's comment, that "human life is driven forward by its dim apprehension of notions too general for its existing language" goes to the crux of the objective-subjective split. It is only now that we can know anything untenuous and certain about that knowledge that we have that we do not know that we have. Since the beginning of recorded time there have been these claims about 'hidden knowledge', about some 'mystical awareness of the connectedness of nature'. But for the fact that some of the claimants have impeccable scientific pedigrees it would be easy to dismiss this material as unsubstantiated nonsense.

However there are people[6] here, of whom it would be hard to accuse of undisciplined ranting. The nub of the problem, then, is the claim that some things about reality, some facts, are simply unknowable in terms of my subjective view of the world. These are the things, like 'truth', 'beauty' and 'virtue', that, in classical Greek thought, were seen as absolute values, but more than that, as the "threefold order of the universe" (Watts 1982, p. 9).

So is it possible to know something and at the same time be unaware that I know it? This sounds like a vicious circle - how can I not know that which I know? This has been the core problem for claims of mysticism all along. To say that there is some fact that it is, in principle, impossible to know, seems to fly in the face of the whole human project. After all this is a much stronger claim than the claim for magic, which is mere illusion - you are experiencing something for which you don't have enough information to explain *at this time*. The claim for the mystical is that, by definition, you can *never* have enough information - here is something that you know, but that you can never explain. However the trick is not that we don't 'know' about these things it's just that we can't *communicate* them. Talking is a function of the conscious mind. I can only talk about that of which I am consciously aware. So the knowledge is there, the information exists in my brain, but I am not conscious of it. It exists in that part of my brain that is not involved in the process of consciousness.

And, at last, science is uncovering hard evidence for the mystical. One example of this evidence is the neurological phenomenon called blindsight, discovered by Larry Weiscrantz and Alan Cowey at Oxford. If the part of the brain known as the visual cortex is damaged, you become blind. For example if the right visual cortex is damaged you become blind on the left side. So, if I pointed a small torch into your left eye and asked you if you could see anything then your answer would have to be 'no'. And if I then asked you to point at it you would ask me the obvious question "how can I point at something that I can't see?" But if I insist, and ask you to just take a guess, then 99% of the time you will 'guess' correctly. That is, you can point to an object that you say you can't see - in fact, you deny all knowledge of it! So the mystics are right, you do 'know' something that you are completely unaware of knowing. The point is that there are different levels of awareness occurring simultaneously in the human brain, fed by different neural pathways from common sensory devices, but the only level that we can talk about is the level that we call 'consciousness' (Ramachandran & Blakeslee 1999, p. 76).

In a sense consciousness is just our interface with the world. There is a lot about us that is, at the very least, a-conscious, if not subconscious. But all the things that make up the particular 'psychological field' of an individual necessarily exist in the individual's mind at some level, they form her "subjective fields of ecological interaction" Uexküll's Umwelt (Conesa 2005). The 'mystical' is simply

[6]Heisenberg, Schroedinger, Einstein, Planck, Pauli, Eddington, De Broglie, Jeans, and the like.

those parts of our make-up, the source of our deepest feelings, emotions, and motivations over which we have little conscious control, but which, nevertheless, are critical to our existence as rational agents.

> There's a whole tradition in the west, going back to Plato; two and a half thousand years of thinking, in which philosophers and later psychologists have regarded emotions as, at best, harmless luxuries and at worst outright obstacles to intelligent action - they get in the way of making intelligent decisions. To use Jon Elster's evocative phrase they're sand in the machinery of action, they just grit things up. And I think one of the most interesting things about the way psychologists and philosophers and neuroscientists are changing their views about emotion over the last, just the last ten years, is that they're realising that this negative view of emotion is fundamentally wrong, and that emotions do sometimes cause us to do things we regret, of course. But if we didn't have them we wouldn't be more rational than we are today, we'd actually be less rational. We need these sort of gut feelings to guide our rationality.
>
> (Evans 2005)

An interesting example of such a-conscious activity is the mysterious workings of the phenomenon we call 'conscience', which, despite our best efforts to 'rationalise' morally dubious behaviour, remains indifferent to conscious manipulation. Yet, despite the lack of conscious access to much of what makes us who we are, we are nevertheless quite aware that we are whole, we construct our conscious awareness of ourselves out of everything that has *meaning* for us, whether or not we can give conscious expression to all of it.

> The human mind is the joiner, fitting together the disparate elements of the world to make objects, systems, sceneries. ...
> What is it about the stuff that temporarily makes up my body, my brain, that places it outside the world of objects and yet gives it the power to draw together objects that are worlds apart? What strange faculty allows me to provide unity and connectedness to objects, where otherwise there would be only timelessness and isolation? How do thoughts, feelings, and our sense of selfhood arise from the machinery in our heads? How does the brain make a mind? And how does the joiner mind link us to the rest of the universe?
>
> (Harth 1993, pp. 9-10)

5.6 Understanding

In a sense this idea that the subjective is just an internal view of the objective world, is the basis of the Theory of Relativity at a lower level. The implications of there being no absolute inertial frame of reference for measuring movement is that there is also no absolute co-ordinate system for the location of objects in space. In thinking about any particular portion of the world I have implicitly set up an arbitrary frame of reference. So, for example, if I am talking about the location of one of the nine planets, then, ultimately, my frame of reference derives from the sun, it being the main determining factor in the structure of the Solar System. The 'order' of the whole system is derived from the gravitational field created by the sun's mass, it drives "local causality" in this part of the cosmos. But a human view of the solar system is conditioned by the fact that humans live on the component of the system called Earth. We therefore have an Earth-centered visual view of the system, a sort of Earth subjective view. Our "difficulty is that we are inside the Solar System, moving along with it, and things that look simple from the outside often look much more complicated from the inside" (Stewart 1995, p. 3). Without some conceptual shift in our viewpoint we can never see the Solar system except in this way. If you go outdoors and watch the movements of the other components of the system over time, it will be your belief that the Earth is the center of the Solar System, and, in fact, of the whole Universe, because what you can only ever perceive with your senses is the fact that everything in the sky revolves around the Earth. The conceptual shift in viewpoint that enables us to see the fact that the sun is the centre, not the Earth, is a shift from a type of subjectivity to a more objective view. It becomes clear, for example, that the apparent motion of everything around the Earth is purely an illusion caused by the viewpoint being within the system. Once you have an outside view the illusion can be seen for what it is.

And the event that triggers the conceptual shift to the different view is understanding. If you look at the world through eyes that have no built in concepts, you just see complexity, a mass of apparently unconnected objects that are all different. Given just this mass of complex detail you can't really understand anything. The first step in understanding is to see beyond the appearance of complexity to the simplicity that underlies it (Stewart 1995, p. 146). So, for example, Ptolemy's explanation of the motion of the planets reflects the observed complexity of the view from the Earth, and Copernicus had to see beyond this complexity, caused by the particular viewpoint that we have on Earth, to understand the underlying simplicity that forms the objective view. The pattern for all the planets, despite the many differences between them, is that they are bodies orbiting the same star, and having the pattern enables you to see beyond the confusion of detail, to understand what is really going on - "The laws of planetary motion become much simpler if this motion is described as relative to the sun instead of relative to the earth" (Weyl 1959, p. 99).

But understanding is, itself, a sort of a trap. After all, the Ptolemists thought that they understood the world. The need for an occasional shift of conceptual viewpoint, or a Kuhnian type "scientific revolution", arises because of the very power of concepts. They are powerful in enabling us to understand the world and we are reluctant to let a particular concept go because the very need to do so says that in some sense our understanding is faulty. We know that relinquishing it will require a lot of hard work to reorient ourselves in the new environment, because accepting it initially reduced complexity, and relinquishing it will take us back, at least temporarily, into a more complex view. But even worse is the fact that our current understanding is a sort of blindness[7]. We can't even see the new way of thinking that is required because the old way colours our view. So, for example, it took us 3000 years to 'discover' non-Euclidean geometry, precisely because Euclidean geometry is a very useful tool in understanding the world (Odenwald 2002). It had helped us deal with complexity in a very fruitful way, and it was only as it began to obscure further progress in this respect that the need to go beyond it arose.

Complexity, the mass of detail that we observe around us, is mainly an expression of differentiation in appearance, and a good deal of scientific method involves classification. The sheer ingenuity of the great 19th Century naturalists who managed to identify a finite number of anatomical archetypes amongst the bewildering variety expressed in different life forms is truly astounding.

> It was from this immense research into basic anatomical types that classical zoology and palaeontology were built - a monument whose structure both evokes and, justifies the theory of evolution. Even so, the diversity of types remained, and it had to be recognized that a great many macroscopic structural patterns radically unlike one another, coexisted in the biosphere. A blue alga, an infusorian, an octopus, and a human being - what had they in common? With the discovery of the cell and the advent of cellular, theory a new unity could be seen under this diversity. But it was some time before advances in biochemistry, mainly during the second quarter of the twentieth century, revealed the profound and strict unity, on the microscopic level, of the whole of the living world. Today we know that from the bacterium to man the chemical machinery is essentially the same, in both its structure and its functioning.
>
> (Monod 1974, pp. 100-1)

This immense effort to classify biological form was paralleled by similar developments at the level of physical material, and again the patterns of form revealed in the periodic table of elements were discovered long before the reason for them could be discerned in atomic structure. It turns out that most of the difference that we see expressed in the world really is just apparent because all the different objects are made from the same basic material, atoms. This was Democritus' great insight[8] - the underlying sameness of everything in the world, despite the

[7]The famous physicist Enrico Fermi achieved nuclear fission in his laboratory four years before it was officially 'discovered' but failed to recognise it because many of his peers, including Einstein, had proclaimed it to be impossible (Shukla n.d.).

[8]Although not the first to conceive of the idea, his was the first attempt at a comprehensive

apparent complexity. So whatever the ordering principle at the atomic level is, it lies behind every single thing in the universe. Let's call it, here, the 'cosmic principle'. In simplistic terms it is ultimately the order that results from the reactivity of nuclei, atoms exchanging electrons[9].

This means that the complexity that we see is just the result of 15 billion years worth of cosmic evolution, of atoms exchanging electrons. It is just different levels of order, physical, biological, social and so on, but levels of order that are themselves based ultimately on the same 'cosmic principle' of subatomic order, atoms exchanging electrons. So 'understanding' is, in a way, just the discovery of the 'cosmic principle'. The result of the 15 billion years worth of cosmic evolution is that the cosmos comes to understand, through that part of itself which is human, the basic 'cosmic principle'. It is order understanding order. If atoms go on exchanging electrons for long enough, it seems, they will become aware of atoms exchanging electrons. The subjective is just the means by which this awareness called 'attention' happens, of making it 'conscious experience'. It is the objective understanding the objective. To see the lion hiding in the bush I have to make 'order' out of a mass of detail[10]. At base, that's all that understanding is - mental order.

The key to the way that humans think is meaning, so the core question of life is 'what does this experience mean in terms of my life?' Meaning derives from reality. It is the underlying order of reality that causes the repetitions that we experience, and therefore order is the basis of the pattern language that we use to understand the world. When you stop to think about it the concept of understanding the world is a very interesting idea. It is what defines us as independent adult human beings. The understanding of the world that we inhabit enables us to make our way, to stand alone, in large part, as individuals, in life. Yet this is not something that anyone gives us, or can give us, directly. It is the product of experience - we acquire the skill bit by bit, in an extremely fits and starts sort of way. What we are given directly by others is knowledge, not understanding, because understanding is mostly a process. Each of us has to fit the bits of knowledge with which we are presented into our own subjective understanding of the world, it cannot come pre-packaged as understanding as such.

---

and systematic view of the physical world based on atoms.

[9]There are complicating factors, such as the subatomic realm, but they don't alter the overall situation in terms of the reactivity between atoms in this context. It just takes the ordering principle to another level. Whatever the order being expressed at the subatomic level is, it is the basis for the atomic level reactions.

[10]Because vision evolved mainly to differentiate between significance and insignificance in detail, detecting the lion hiding in the bushes, handling visual form is critical. The scene, in detail, is just a vast mixture of irregular green and yellow bits. What our vision has to do is to put the yellow bits together despite all the green 'noise'. The form behind all the apparently unconnected yellow bits is 'lion', and that's what we need to see. We 'order' the scene to discover the significance of the yellow bits, to understand the danger. Without a pre-existing concept for 'lion' seeing the danger would be impossible.

Yet although the process and its result can be seen as being entirely subjective, the understanding involves, indeed implies, a correspondence with the real world. An attribute of understanding is that it enables one to deal with reality successfully and usefully, otherwise it is its opposite, misunderstanding. That is, if the process produces an understanding that is too strongly based on subjective opinion, wishful thinking, then it will hinder rather than help a person make their way through life. So although understanding is, in part, a process that takes place in an individual mental subject, it is conditioned by having to contribute to the subject's survival. So this is the equivalent, in terms of the individual, to biological evolution, with the added complexity that the environment to which the subject is being fitted contains a large component that is mental rather than physical in nature. We are biological beings and as such we have to survive in the physical environment. The human mental component is, first and foremost, a factor in our continuing survival.

This implies, as Piaget points out, that, just as biological acts are acts of adaptation to the physical environment and organizations of the environment, so too are cognitive acts. The overall process by which an organism adapts to the environment and organizes experience consists of both mental and biological activity. Intellectual functioning is a special form of biological activity (Piaget 1952, 42). The intellectual development of an individual can be seen as a series of acts of organization of and adaptation to the perceived environment. This doesn't mean that there is a direct relationship between mental and biological functioning, only that these concepts from biology are applicable to studying intellectual development. It is this continuing project to understand reality that drives the human condition, which can be seen both as the internal mental world and the physical world as modified by our mental activity. The two aspects go together to define the human condition. As we accumulate experience the initial pattern language that we learn as children differentiates into many different, but overlapping, pattern languages, one for each discrete category of experience.

In this sense programming is no different from any other aspect of human effort - as an attempt to 'make sense' of an environment in which problems are solved by means of computer programs, it will be based on a pattern language. The process of leaning to program is simply the process of learning the appropriate pattern language. We believe that the reason that many people have trouble in learning to program is because this essential point is missed by most instructors. Instruction is presently based on an artefact that is called a 'programming language', but which is not, in fact, a true language at the lowest level in the way that a pattern language is. It is designed basically for communication, in this case, communication with a compiler, not, primarily, to provide conceptual understanding in the human using it. Basing instruction purely on such a language is highly dangerous because of its inherent inflexibility. Natural language is highly flexible, we don't normally misunderstand people simply because they use a badly formed sentence - meaning overrides syntax. At least part of the difficulty that novices have with learning to program is due to the inflexibility of the programming language. It fits the way that the compiler works not the way that humans think about problems.

In this regard there has been an ongoing saga surrounding the activity of programming ever since the beginning of the computer age. From the start it has been seen as a difficult art and a lot of the subsequent development of computers has been driven by the desire to make using computers easier. The first stage was the breaking of the nexus between using and programming that existed in the beginning.

> When Weinberg wrote his early monograph "The Psychology of Computer Programming", it was assumed that every serious user of a computer would be a programmer. He used the term 'programmer' almost as a synonym for user (some people performed mundane operations, such as data entry or job control - but these existed solely to serve the programmers). It was the development of standardized packaged software ... that led to classes of professional computer user who were not programmers.
>
> (Blackwell 2002, p. iii)

In terms of the programming strand, the continual thrust for higher level programming languages can be seen as a quest to bring the cognitive tasks involved in programming closer to our normal ways of thinking. "Programming ... is basically a process of translating from the language convenient to human beings to the language convenient to computers"(Blackwell 2002, p. i). This convergence was the stated goal of the developers of Simula, in 1967, the first attempt at an object-oriented programming language.

> The basic philosophy underlying the Object-oriented programming is to make the programs as far as possible reflect the part of reality which they are going to treat. ... The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.
>
> (Holmboe 1999)

In a sense, the move to Object Oriented programming is a step in the direction of rationality from formal logic predicated on the basis that "a purely logico-deductive attitude is not natural to the human mind" (Weyl 1959, p. 19). Rationality is often, if only implicitly, taken to be equivalent to formal logic but this is not the case.

> The trouble with identifying rationality with the formal canons of logic simpliciter is that rationality is a broader and more complex notion than logicality. Rationality is tied up with issues related to the meaning and quality of our individual lives in a way logicality is not.
>
> (Flanagan 1991, p. 209)

Simula and its descendants are thereby an attempt to address the difficulty that talking directly to a computer system requires thinking in formal logic rather than thinking in the normal human sense of the word. There is a direct mapping between the formal logic that underlies assembly languages and the physical circuitry of the computer that doesn't exist in ordinary human thinking.

As the book, "Psychology of Programming" suggests, programming has evolved from "describing calculations", through "defining functions", to "defining and treating objects" (Blackwell 2002). All this activity in programming language development is predicated on an implied correspondence between programming and problem solving (Bergin et al. 1997, p. vii). A programming *paradigm* is equivalent to a different problem-solving philosophy, one applies one's 'native' skill in solving problems to the design of classes that describe objects, in the case of Java or C++, instead of procedures, as in Pascal, or functions, as in Haskell. But, despite these attempts to bring programming languages closer to normal thinking, there still remains the basic difficulty of teaching people how to perform this seemingly esoteric art. And a lot of the difficulty does, in fact, seem to derive from the fact that programming is indeed an art, in the sense of requiring a degree of creativity. At the fundamental level, "discovery is art, not logic" (Smith 1981, p. 347). So the difficulties that arise in the learning to program situation are a continuation of those expressed by educators in other fields where thinking in creative ways is required. It might be that trying to think in too logical a way inhibits the free flow of imagination that underlies creativity.

5.7 Creativity

One of the main problems in dealing with creativity is the tendency to see it as an innate factor, not something that can be explicitly taught. This attitude leads to a radical neglect in training for creative expression. But it is clear from the playful creativity of children that we all have a measure of it. As Picasso once said, "every child is an artist. The problem is how to remain an artist once we grow up." This suggests that maturity brings with it an inhibitory impulse that stifles the spark of creativity that lies within us all. Some of this is just the pressure of time and responsibility but it is likely that the loss of the lack of the fear of failure that children exhibit is also a factor here. Children have a complete indifference to failure that is demonstrated, for example, in their efforts to learn to walk. Despite failing many times it never causes them to stop trying. Somewhere along the road to adulthood we teach them to fear failure.

> As a general rule adults give up trying new things after the first or second attempt, and it is a rare individual who continues again and again after repeated failure. Yet we are all aware of those rare people who practice something over and over again, in such diverse fields as sports, music, literature or business; and they sometimes achieve

> greatness. In their case they simply adopted the drive of the child they once were and avoided the trap of fearing failure.
>
> (Kaipa & Johnson 1999)

It is made easy to believe that creativity springs from some individual trait because of the situation that pertains in our society. When you think about poetry, for example, it looks like poetic ability in an individual springs from nowhere, as, at the educational level, there is very little in the way of explicit training in poetic skills, in fact, for all intents and purposes, none at all. About the only exposure that most people have to poetry is those poems that they are induced to read as part of the school curriculum. And, indeed, for most of us this limited exposure is seen as a negative experience, so that if, indeed, we had any 'innate' poetic ability, the 'force-feeding' aspect of our school experience would very likely tend to dull rather than nurture it. Given this complete absence of training or motivation it is amazing that poets still do arise from such infertile ground. So the general social situation surrounding poetry in our society virtually forces us to see it as something in the nature of the individual personality of the poet rather than a 'skill' that has to be acquired through practice.

But it is clear that any creative activity occurs in a material context; it deals with real entities, activities and relationships, even though they might be, as in the case of poetry, completely abstract. So although there is undoubtedly an element of individual talent or aptitude involved this can't be the whole story. When one looks at the process of creation itself it is clear that, mostly, one is using aspects of everyday thinking like classification, generalisation, abstraction, and so on. The most outstanding feature of the creative process is that it is multi-faceted, which suggests that the real problem in addressing it in educational situations is the logistical complexity of the situation in which it is occurring rather than any subjective qualities that might or might not be innate.

> Many creative products in the real world - a poem, a scientific theory, an innovative business - are extended products, involving many parts and aspects. Also ... defining the problem initially is an important part of the effort, and problem finding abilities seem to bear a close relation to creativity. Notable creative efforts appear to involve an element of abstraction: one constructs a product based on general and far-reaching considerations. And, of course, there is no guarantee of a satisfactory solution when one addresses a real world problem. Finally, most often there is no guide through the complexities of a realistic creative challenge. One is on one's own, or working with a group equally in the dark.
>
> (Nickerson et al. 1985, p. 213)

Although problem solving is a creative activity, it still involves thinking logically, and this too, is often regarded as a characteristic of the individual. However, John Stuart Mill, who is generally considered to be one of the keenest minds the world has produced ((Nickerson, Perkins & Smith 1985, p. 37) and Ruth Borchardt in (Mill 1946, p. 7)), attributes his abilities entirely to the training he received as a child through the efforts of his father. In his autobiography he describes his experience as an experiment which "proved how much more than is commonly supposed may be taught, and well taught" (Mill 1981, p. 5), and goes on to discuss his own case in some detail.

> The result of the experiment shows the ease with which this may be done ... If I had been by nature extremely quick of apprehension, or had possessed a very accurate and retentive memory or were of a remarkably active and energetic character, the trial would not be conclusive; but in all these natural gifts I am rather below than above par; what I could do, could assuredly be done by any boy or girl of average capacity and healthy physical constitution: and if I have accomplished anything, I owe it, among other fortunate circumstances, to the fact that through the early training bestowed on me by my father, I started, I may fairly say, with an advantage of a quarter of a century over my contemporaries. ... Mine ... was not an education of cram. My father never permitted anything which I learnt to degenerate into a mere exercise of memory. He strove to make the understanding not only go along with every step of the teaching, but, if possible, precede it. Anything which could be found out by thinking I never was told, until I had exhausted my efforts to find it out for myself.
>
> (Mill 1981, pp. 33 and 35)

Mill's testimony suggests that the abilities like understanding and thinking logically are acquired through training rather than being a characteristic endowed on the individual. The goal of disciplined and logical thinking is essentially that of philosophy, and the Socratic method explicitly aims at developing the style of thinking used in analysing and solving problems (Tekinerdogan & Aksit 1999). Aristotle developed the idea of logic from the Socratic dialogue. It is based on three levels or stages (Rayside & Campbell 2000). The first stage is based on defining the entities in the area of interest. It mainly involves identifying and categorising objects according to their independent existence. Stage two is the characterisation of the entities in terms of the attributes or properties that they display. These first two stages set the scene for the real action, the discourse, or argument, by which further knowledge is derived from that which is already established. Basically this is, coming to know the reason for the system under study being the way that it is and the causes behind its current state; it is this factor that gives this stage its name, reasoning. In a sense, then, logic, as it was outlined by Aristotle, is the mental mechanism that underlies the Socratic method by which we come to understanding.

Aristotelian logic is a thereby a formalisation of rational thought and is unfortunate in some respects, particularly in the separation of the 'rational' and 'poetic' sides of language that it implies. In this connection it is interesting to consider the roots of these two words. The word, 'rational', derives from the root, 'ratio' and suggests a correspondence, 'a ratio', between the entities in our mental representation and the entities in the real world that they represent. In other words rational thought is the type of thinking that preserves, or, at the very least, tries to preserve order across the subjective-objective divide. The order that we perceive in the world is reflected in the way that we think about it - objective order becomes conceptual order, understanding.

The root of the word 'poetry' is the Greek 'poiein', meaning 'to make' or 'to create', and this is interesting in that it applies to creativity generally, not specifically to creative expression in language. So the essential difference, in terms of these two styles of thinking, is creativity, and the fact that only one of these styles has ever been formalised, suggests that their separation carries disadvantages as well as the obvious advantages in pragmatic and even survival terms. C.P. Snow's famous 'two cultures' division derives from this divergence, but the consequences of the split at the individual level might be even more profound. If, as we are arguing here, understanding is a sort of creative act, the separation of rational from poetic thinking implied by the formalisation of the former into logic obscures this essential point. Of course, the point that the philosophers were making was that analysis underlies understanding, not that analysis *is* understanding, but this point is easily lost in the flow of formal logic.

> The proper order of operation of the mind requires an overall grasp of what is generally known, not only in formal, logical, mathematical terms, but also intuitively, in images, feelings, poetic usage of language, etc. (Perhaps we could say that this is what is involved in harmony between the 'left brain' and the 'right brain'.) This kind of overall way of thinking is not only a fertile source of new theoretical ideas: it is needed for the human mind to function in a generally harmonious way, which could in turn help to make possible an orderly and stable society.
>
> (Bohm 1980, p. xiv)

5.8 Logic and Understanding

Thinking clearly requires logical progression, but its end is understanding, which involves novelty, and, therefore, creative thinking. So this is a very old theme, and the main difficulty has been in discovering ways to orient educational programmes towards the fostering of the skill of thinking in its totality rather than the mere installation of knowledge through the straight presentation of facts. As Edsger Dijkstra has said "one of the main objects of education is the insight that makes quite a lot of knowledge superfluous" (Dijkstra 1982, p. 38). This idea that education is mainly about the fostering of thinking skills reaches back to the beginnings of the modern mass education system. In fact it was the professed aim of the Latin School movement, which continued in the use of Latin long after it had ceased to be the language of commerce and even the principle language of religion[11], to inculcate proper "habits of mind" (Rippa 1967). So the justification for this concentration on Latin was that learning it was thought to 'train the mind', that is, that it was useful in terms of developing thinking skills rather than in knowledge of the language per se.

As the scientific revolution unfolded, mathematics and physics came to the fore as prime candidates for the role of training in the art of solving problems. But educators in these fields have long reported on the difficulties involved in teaching the art. And, of course, this raises the issue about what it is that is actually being taught. It has been said of the education system that it doesn't teach students how to think, that it requires them to think without ever teaching them how (Halpern 1987, p. 75), that they are presented with the material on the assumption that they already know how to think. This points to the notion of different types, or levels, of 'thinking', and the suggestion is that the fact that people can demonstrate facility in one type of thinking, that involved in everyday activity, does not mean that they will be able to handle the more abstract problem solving involved in these specialised fields without explicit training.

However, in order to deal with this issue, we need to know in what ways the thinking involved is different in character to that of the everyday thought processes that everyone exhibits. Everyday activity revolves around the process of living, of organising the things that need to be done. On the face of it, the creative thinking required to solve a problem in the abstract would not appear to be all that much different, you still just have to figure out what needs to be done, and how to do it. But it is clear from all the reports that the normal type facility of everyday thought does not transfer directly into thinking in more abstract fields. Education cannot simply devolve to providing the knowledge required to think creatively in a field, it is clear that there also needs to be explicit training in its use in solving problems, more than just presenting examples.

The classic model for the training of a particular skill is the apprenticeship model, whereby the novice works directly with the expert. This seems to work well for skills that require manual procedures, where all the aspects of an activity can be directly observed, but mental activity is not visible for inspection in the way that manual activity is. Nevertheless it would appear that the fundamental premise of the Socratic method is that the teaching of thinking skills can be done in this way by exposing the thought processes involved through discussion. The expert presents the novice with a problem and by continually questioning the student causes her to justify every mental move made along the way to the solution. Plato's dialogues clearly illustrate this mode of mental training, and

[11] "For nearly a thousand years after the fall of the empire, Latin continued to be the language spoken in commerce, public service, education, and the Roman Catholic church. Most books written in Europe until about the year 1200 were written in Latin." (Guisepi n.d.)

the question becomes how can it be incorporated into an educational system that has to deal with large numbers of students where the one-on-one apprenticeship model seems hardly possible?

The default approach seems to have been the one that Halpern points out - you assume that students already have the skill of 'thought' and that it will transfer automatically to the new material (Halpern 1987, p. 75). A second common strategy based on the assumption of a transfer of skill in one area to another is the teaching of a particular subject or subjects in the belief that they will give students general thinking skills - the theory behind the Latin School movement, and more latterly the teaching of mathematics. These methods are leavened by the adding of practice sessions whereby novices solve given problems under supervision. What becomes critical in all such approaches are the assumptions being made. These are things like the idea that mental skill does 'transfer' across fields and that knowledge is sufficient condition for this, that thinking can be treated generally, both in terms of how individuals think and how it applies in different areas, and even that it is something that can be explicitly taught.

These assumptions all come together in the push to introduce general 'critical thinking' type courses into the education system. This raises the thorny issue about what 'critical thinking' actually is, so that you can teach and assess it.

> Critical thinking and problem-solving courses are predicated on two basic assumptions: (1) that there are clearly identifiable and definable thinking skills that students can be taught to recognize and apply appropriately, and (2) that if these skills are recognized and applied, the students will be more effective thinkers. A list of such skills typically would include understanding how cause is determined, recognizing and criticizing assumptions, analyzing means-goals relationships, reducing complex problems to simpler ones, making appropriate inferences, and ... using analogies as an aid to comprehension, memory, and problem solving.
>
> (Halpern 1987, p. 75)

But the fundamental assumption here is still the generalisation of thinking issue, the notion that problem solving can be treated in isolation in this way, that it is not something that is dependent on a particular context. So we need to be clear about what we think the problem solving process really is before we even think about how to teach it to novices.

As stated earlier, the tendency has been to bring the programming environment closer to the way it is assumed we think normally, on the basis that we have great difficulty in handling the pure logic of the computer. In other words, we have tried to move towards the less rigid human style of logic. At the programming language level this is constrained by the fact that a program still has to be translatable into pure logic, so the trend has concentrated, in the main, on representational factors, the treatment of the problem domain, and on facilitating comprehension through the use of more natural language mechanisms. This would all seem to be predicated on the assumption that writing a program is largely analogous to general problem solving.

Traditional logic is a way of reasoning about reality. It requires a fundamental correspondence between the symbols being used in the mind, and the aspect of reality that they represent. But this is not the logic that is being used inside of a computer. Computer logic, as pure symbolic logic, requires, indeed, no such correspondence with reality, its power derives from the fact that it is totally abstract. A 'truth' is simply a value in symbolic logic, a product of logical sequence, not a literal truth in the sense that it represents a fact about the real world. So the task of programming can be seen as the act of translation between the human and symbolic forms of logic. Symbolic logic is how the computer operates, but not how the programmer thinks (Rayside & Campbell 2000). Fortunately, most of the work that is done by computers involves aspects of the real world, the problems being solved are 'real world' in origin, so the fact that they are ultimately being dealt with in a process that is purely symbolic is a nuisance not a critical stumbling block. The push to move programming practice closer to human thinking patterns would therefore seem to be soundly based. What is needed is an appreciation of reasoning as 'systematic' thinking (Hauck & Freehill 1972, p. 4), thinking directed at building an understanding of whole systems rather than thinking based on proving concepts formally.

It seems that in the development of programming we have repeated the mistakes of traditional psychology whereby process is separated from context.

> The missing relationship between thought and reality is 'meaning'. Thinking is dissociated from the environment in which it is taking place. When we approach the problem of the interrelation between thought and language and other aspects of mind, the first question that arises is that of intellect and affect. Their separation as subjects of study is a major weakness of traditional psychology, since it makes the thought process appear as an autonomous flow of "thoughts thinking themselves," segregated from the fullness of life, from the personal needs and interests, the inclinations and impulses, of the thinker. Such segregated thought must be viewed either as a meaningless epiphenomenon incapable of changing anything in the life or conduct of a person or else as some kind of primeval force exerting an influence on personal life in an inexplicable, mysterious way. The door is closed on the issue of the causation and origin of our thoughts, since deterministic analysis would require clarification of the motive forces that direct thought into this or that channel. By the same token, the old approach precludes any fruitful study of the reverse process, the influence of thought on affect and volition.
>
> (Vygotskii 1962, p. 10)

A similar context-shifting phenomenon is apparent in programming. The only way that a programming language construct can be understood, at the level of the programming language, is by apprehending its effect during execution, as that is the 'meaning' that it has at this level[12]. "Suggestions for making a program more reliable implicitly embody, or define, notions of meaning. Thus, in order to prove that a program always behaves as it should, it is first necessary to define 'behaviour', and this is essentially a commitment to a definition of meaning" (Brady 1977, p. 218).

But appreciating the effect of a machine instruction means simulating in the mind the activity of the machine. This is simply an endeavour in which normal everyday experience has not given the human mind much instruction or practice and it is this fact that causes bugs in programs and which makes debugging such a difficult exercise, so much so that we use automated execution tools, 'debuggers' or 'simulators', to help us find them, suggesting that analysing logical expression for correctness (Brooks 1983, p. 65) is best done by machine rather than mind. We are simply not used to analysing a situation in terms of strict logic[13]. The fact is that we are more used to thinking in terms of language than logic. So, although the means of communication between programmer and machine is called a 'language', it is, in fact a system of logic. We are misled if we take the 'programming language' metaphor too literally - "the metaphor is more misleading than illuminating" (Dijkstra 1982, p. 290) - this is an altogether more rigid environment than normal language. In thinking about programming purely in programming logic terms we separate it, to some degree, from the real life problem we are trying to solve. Pattern language, that is, the language we use in dealing with real life, the "effective ordering of one's thoughts" (Dijkstra 1982, p. 62), is our primary skill-set so there are advantages in attempting to make the medium of our problem solving in programming closer to pattern language than machine logic. As Stephan Somogyi has said "programming is never going to be easy, since it forces otherwise sane people to think like computers." (Somogyi 1999). We need to enable programmers to think like humans, not machines.

The trouble is that we are not yet clear about the process of human thinking itself, let alone how to apply it to the task of writing programs. Moreover our concern here is with yet another level of mind activity, learning.

> When we refer to learning, we usually relate only to our conscious mind. Our consciousness is the familiar part of our thinking that

[12]"In Departments of Computing Science, one of the most common confusions is the one between a program and its execution, between a programming language and its implementation. I always find this very amazing: the whole vocabulary to make the distinction is generally available" (Dijkstra 1982, p. 64).

[13]The same problem occurs when comparing two computers for software compatibility. It is simply too difficult to do it by comparing the manuals of both, the only way to tell if a new machine 'matches' an existing one turns out to be to "ask the machine". Test programs are written to determine the behaviour of the existing machine, and the new machine built to match (Kidder 1981, pp. 64–65). This procedure is extensively outlined in Tracy Kidder's "The Soul of a New Machine" describing the construction of Eagle, a new 32-bit mini computer that would be "*fully* compatible" with Data General's previous machine, the 16-bit Eclipse (Kidder 1981, p. 46)

> we use all day long, and because of this it is easy to assume it is our only information storage device. However, our subconscious mind learns much more than our consciousness, and it does so seemingly effortlessly. ... For example, its lightning-fast ability to recognize the potential for metaphor in a totally new stream of information, by comparing it with a single experience of perhaps thirty years previous, probably means that it may actually think in metaphor and profound patterning in the first place. This is then interpreted by the conscious mind and translated into our language as needed. Such concepts as the English Language, arithmetic, philosophy and logic may be mere intellectual devices created by the conscious mind to improve functioning and to better feed new pattern sources down into the ever hungry unconsciousness, where deeper and more potent thoughts can simmer and evolve in order to send waves, trends and hints upwards again.
>
> (Kaipa & Johnson 1999)

So before we can even think about teaching people how to write computer programs we have to understand the program writing process itself. To do this we have to match the skills required for the programming task with the cognitive resources available to the human mind[14]. Which aspects of human thought come into play during the design of a program and, in particular, in learning how to program?

5.9 Language and Understanding

It seems to us that the answer is clearly based on the idea of a pattern language. The skill that we demonstrate in understanding the world, and the ease with which we develop that skill, indicate that here lies the answer to the dilemma of the teaching of programming. Conceptual understanding is our core competency. So the truth is that programming skill *should* derive from general thinking skill; the fact that it doesn't tells us that, as educators, we are not presenting the material in a way that addresses our primary skill.

> When study after study, regardless of the methodology, reaches the same conclusion, there must be some common, underlying phenomenon that is being measured. An example in point is the large number of studies concluding that novice programmers know the syntax and semantics of individual statements but they do not know how to combine these features into valid programs.
>
> (Winslow 1996)

The phenomenon that is being measured here, we believe, is the disjoint between the way that the human mind normally works and the formal logic that

[14]This is discussed in Chapters 8 and 9

underlies the programming language. Knowing the rules of logic, in this case the programming language statements, is not enough to guarantee skill in applying them.

> [We know that this is so from studies that show] that even trained logicians sometimes have trouble applying such standard rules of inference as modus ponens (if A implies B and if A is true, then B is true) and modus tollens (if A implies B and B is false, then A is false). They get confused by quantifiers ("all," "some," "each") and they have trouble processing negative sentences. These data undermine any naïve platonic confidence that all people need is a gentle academic reminder of the deductive logical principles that they already, in some sense, "know."
>
> (Flanagan 1991, pp. 210-11)

Novices *know* the programming language but this is patently not enough to make them competent in its use. What we need to base our teaching on is language in the widest sense, language-as-understanding, not a language primarily designed for communication with a compiler. Studies of learning strategies from experiments on the learning of artificial languages (Carroll 1995) suggest that the semantic content of a message is more important in learning than formal aspects of language. So it is conceptual understanding that underlies programming skill, not simply the knowledge and naïve use of programming concepts.

> [As instructors we] often fall into the trap of jumping into code examples far too quickly. These are primarily concerned with the way the language is actually written, its syntax. ... [Compilers and] interpreters are already very good at determining when a programmer has mis-spoken in their program. Syntax errors cannot be bypassed; programmers new to the language will eventually learn through experience what the right thing is in terms of syntax through trial and error, but the concepts are usually what really bites someone.
>
> (Lee 2003)

Of course these elements of syntax will be present in the pattern language for the teaching of programming but it is a language designed for understanding not just communication and is therefore not constrained by syntactic absolutes.

Programming languages are, in essence, pure syntax - they consist of the building blocks of expression only. This fragmentary approach is, of course, necessary for the process of programming, but it leaves the creative aspect of expressiveness as something that is fundamentally mysterious. Our field is not unique in this respect, this tendency to fragment knowledge and to leave creativity as an 'aptitude' of each individual is a feature of all scientific endeavour.

> The prevailing tendency in science [is] to think and perceive in terms of a fragmentary self-world view ... Such a way of thinking and looking in scientific research tends very strongly to re-enforce the general

> fragmentary approach because it gives men a picture of the whole world as constituted of nothing but an aggregate of separately existent 'atomic building blocks', and provides experimental evidence from which is drawn the conclusion that this view is necessary and inevitable. In this way, people are led to feel that fragmentation is nothing but an expression of 'the way everything really is' and that anything else is impossible. ... One might in fact go so far as to say that in the present state of society, and in the present general mode of teaching science, which is a manifestation of this state of society, a kind of prejudice in favour of a fragmentary self-world view is fostered and transmitted (to some extent explicitly and consciously but mainly in an implicit and unconscious manner).
>
> (Bohm 1980, p. 15)

In terms of providing manageable sized chunks of information this is a powerful and essential tool. However designing a program requires putting the pieces together and programming languages, in themselves, provide little in the way of assistance in this regard.

The differences between language-as-conceptual-understanding and language-as-communication seem to lie at a very fundamental level similar to the distinction that David Bohm makes between the flow of meaning and the rigidities imposed by language structure. In order for meaning to be expressed completely freely any imposed division is likely to be counterproductive. That is, there are rigidities imposed by the use of words. Ideally, meaning should just flow. But words demand a degree of 'atomisation' of ideas. In order for communication to work there needs to be a conventional aspect - units of 'agreed' meaning - and this causes problems.

> [The flow of meaning] is rather similar to that of field theory in physics, in which 'particles' are only convenient abstractions from the whole movement. Similarly, we may say that language is an undivided field of movement, involving sound, meaning, attention-calling, emotional and muscular reflexes, etc. It is somewhat arbitrary to give the present excessive significance to the breaks between words. Actually, the relationships between parts of a word may, in general, be of much the same sort as those between different words. So the word ceases to be taken as an 'indivisible atom of meaning' and instead it is seen as no more than a convenient marker in the whole movement of language, neither more nor less fundamental than the clause, the sentence, the paragraph, the system of paragraphs, etc.
>
> (Bohm 1980, p. 41)

If anything, programming languages are even more rigid, in this respect, than natural spoken languages. Conventional aspects are even more tightly enforced. The style of language that is most 'free' in this regard is thinking because there is none of the ‘common denominator’ aspect of communication involved. It doesn’t matter if I have an idiosyncratic understanding of a particular concept, until, that is, I attempt to communicate with others. In this sense pattern languages are constrained only by the ‘meaning’ relationship between a recurring form and its representation in the mind. This is probably the closest that one can get to a ‘free flow of ideas’, as it is basically a flow of associations, where the association is between the mental representation and recurring form. It doesn’t matter at what level of order or understanding that a repetition occurs, if it repeats then I need an appropriate mental representation - a pattern. So the fragmentation into units of meaning is not constrained by convention, the need for agreed meaning, but by the recurrent experience directly.

Learning to program is, in some ways, similar to learning to speak, and yet in other ways quite different. As educators we need to be clear about these similarities and differences in order to make the process as easy as possible. Speech occurs in a manner that obscures the fact that there are at least two levels of activity occurring, deciding what you mean, and translating that meaning into words - conceptual design and implementation. These two stages appear to occur simultaneously and the ease with which it happens in speech has probably contributed to us tending to overlook the fact that there are these two stages in language use when we come to thinking about programming.

A child acquires its first spoken language one word at a time, first by simple mimicry of what it hears and later by explicitly asking for the word that represents various items in its world. Mostly, though, the learner comes to pick up new concepts merely by hearing them used several times, that is, they are ‘discovered patterns’. Meaning is the key to learning the use of a new concept and the critical factor for the learner in discovering meaning is context. The way that the concept is used in speech gives the listener most of the information needed to ascertain its meaning. Often that is all the learner needs to assimilate a new word, but it can always be checked by asking a person more experienced in speaking the language, or later in life, of course, by checking it in a dictionary.

When a child speaks its first sentence its useable vocabulary, if ‘usable’ is defined in terms of what it is capable of putting into sentence form, is, most probably, only those words actually used in the first sentence. It will, of course, have a larger vocabulary consisting of other words and phrases that it has learned, but mostly these are, as yet, single uncombinable concepts for the child. One could say, then, that natural language acquisition is a staged process of linguistic organisation driven by context. A pattern language for programming needs to be presented to novices in a way that mimics this natural language acquisition process as closely as possible, that avoids setting up in the novice a vocabulary of single uncombinable concepts. What the novice needs most is a useable language - what we give them is a list of dictionary definitions, and, not surprisingly, confusion reigns.

For the first program presented to a novice, a pattern language encompassing the bare minimum of concepts needed for that program should be used. As further programs are developed any new concepts needed for a particular program are added as patterns to the language. This way, the novice programmer, like the child learning to speak, has only to deal with new ideas as they are needed, that is, entirely in context. Concepts are 'discovered' through use, not just presented in isolation as 'facts' to be learned. Language is acquired 'actively', not 'passively', it is learned by using it. This process is illustrated in Section 9.4.

No child learns to speak through a process of being presented with a 'complete' vocabulary separate from use, that is, largely isolated from context, yet that is how, more or less, we expect novice programmers to learn. We present them with the set of programming language constructs in the abstract and expect them to derive the most important aspect of their use, context, from a few forlorn examples sprinkled throughout the teaching material. This is about as far from normal language acquisition as it is possible to get. In terms of novices a pattern needs to be written in a way that encompasses all the information needed to use the programming language concept that it is dealing with except context. Context should be dealt with in the pattern language diagram. This is important because in designing a program you should, ideally, be dealing only with the concept, not the details of its code form. The pattern form, in other words, hides the coding details while you are working on the conceptual design of the solution, that is, while you are using the pattern language diagram. A lot of the difficulty that people experience comes from trying to do the conceptual design of a program at the same time that they are wrestling with the implementation details.

This is where learning to program differs from learning to speak. When you are speaking, conceptual design, that is, the meaning that you are trying to express, will more often than not override implementation factors. People will understand you even if you mess up the syntax, or express your ideas ambiguously. This is not the case with a computer program, the conceptual design has to be perfect. So the best way to facilitate the use of patterns in the conceptual design stage is by means of a diagram of the contextual relationships between them - a semantic diagram of the language. Thus when you are designing, all the information you need for construction of the conceptual framework is encompassed in the diagram. As with a new word you might occasionally need to confirm your understanding of a particular concept by checking with a dictionary, in this case the detail contained in the pattern form, but this will probably be rare, especially if the programs presented to the novice are wisely chosen so that each successive version of the developing pattern language contains few concepts new to the programmer.

The idea of basing the learning of programming on a pattern language, therefore, provides advantages in the following ways. Firstly the patterns act to separate the coding process from the activity of design. The coding details are still all there, but they are hidden inside the pattern form. Secondly, the pattern language is a map of the contextual relationships between the patterns and this gives the novice much closer direction than is provided by raw programming language syntax[15]. Thirdly, the pattern language provides the basis for an evolutionary understanding. New concepts are not just presented in isolation, but are fitted into an existing context in the language diagram. This explicitly mirrors the way that we normally acquire new information as we progress through levels of understanding. New concepts, as they are learned, are fitted into the existing mental picture, the cognitive structure in Piaget's terms[16], and complex concepts will usually cause a shuffling of the current understanding. Complex concepts form small pattern languages in their own right, so incorporating them involves a merging of two pattern languages[17], which usually causes changes in the structure of the original language. But the main advantage of a pedagogy based on a pattern language is that it is built on our primary competency as human beings, understanding the world.

Fortunately for us, the way that the world is makes understanding it a feasible proposition. It is an ordered system and it is this factor that underlies the patterns that we perceive. The relationship between the pattern and the way that we represent it in our minds is meaning - meaning is just the significance of real world phenomena to us. So this is a semantic system - a language. It is a language that is used to build understanding of the real world in the human mental system. The human project can be seen as the evolution of a mental system to understand an ordered system, reality. Programming is just a means of dealing with another ordered system so it too needs to be based on a language for understanding - a pattern language. Teaching people to program would be a lot easier if it used an approach that directly addresses the primary human competency, understanding things.

> The most important outcome for ... instruction is the student's learning with understanding. If something is not taught so that it can be learned with understanding, then instruction should be changed so that understanding can take place; or teaching that content should be postponed; or, if that content can't be taught meaningfully, then it shouldn't be taught at all. ... Understanding means more than that students simply display what they have just been taught. ... Understanding means that students can link what they are learning to previous knowledge that they already (should) have. Understanding means that students can explain why they believe something is true in a way that is sensible to someone else.
>
> (Secada & Carey 1990)

It is often forgotten that programming is a mixture of science and art. The programmer is dealing with concepts at the level of science but combining them

[15]This sequencing process is discussed in Chapter 7

[16]"Every schema is ... coordinated with all other schemata and itself constitutes a totality with differentiated parts."(Piaget 1952)

[17]This merging process is discussed in Section 7.6, and further in (Porter et al. 2005).

to achieve novel goals - each program is, in a sense, a new creation. Of course this duality has always been true of the scientist in general but scientists have tended to downplay the artistic and even playful aspects of their work. “It is high time that scientists admit that their experience in the laboratory is an aesthetic one, at times acutely so: the arid form of presenting their results has disguised this, and their respectable logical front often makes it invisible even to a student” (Smith 1981, p. 234).

Any human activity is both scientific, in the sense that it is building on what is known, and creative, in that it is directed towards achieving something new. Even the sudden flash of insight that we call inspiration must be preceded by thought of some kind, unconscious though it may be. It is unlikely to be the case that it has arisen entirely spontaneously in the mind.

> Consider a musician who faces an impasse while composing a score. Sometimes, the solution will come as if from ‘nowhere’. But clearly, the unconscious mind accessed the relevant representations to solve the problem. Here, there is presumably attentional access to sensory representations, and cognitive access to musical theory and aesthetic standards. But the representations were unconscious while the problem was being solved.
>
> (Weisberg n.d., p. 91)

Some language-like process involving the combination of concepts into new form is occurring somewhere in the brain, and is springing unbidden into consciousness as a flash of sudden understanding[18]. Language’s end is meaning, and it is only in terms of meaning that it can be creative.

[18] “To see the reason for something is not a mechanical activity ... Rather, one is aware of each aspect as assimilated within a single whole, all of whose parts are inwardly related (as are, for example, the organs of the body). Here, one has to emphasize that the act of reason is essentially a kind of perception through the mind, similar in certain ways to artistic perception, and not merely the associative repetition of reasons that are already known. Thus, one may be puzzled by a wide range of factors, things that do not fit together, until suddenly there is a flash of understanding, and therefore one sees how all these factors are related as aspects of one totality (e.g. consider Newton’s insight into universal gravitation). Such acts of perception cannot properly be given a detailed analysis or description. Rather, they are to be considered as aspects of the forming activity of the mind. A particular structure of concepts is then the product of this activity, and these products are what are linked by the series of efficient causes that operate in ordinary associative thinking” (Bohm 1980, p. 13).

Chapter 6

Pattern Language Fundamentals

Repetition is the only form of permanence that nature can achieve.
George Santayana

It will not now be necessary to prove, that those perceptions, which are simple, and exist no where, are incapable of any conjunction in place with matter or body, which is extended and divisible; since it is impossible to found a relation but on some common quality.
(Hume 1740)

6.1 Connectivity and Creativity

The combination of concepts into new form, creativity, is the process that is driven by pattern language. To see how this works we need to understand the connections between the recurring features that exist in the real world and the conceptual process. In essence, the pattern language idea boils down to a way of looking at a system. A system is a collection of related concepts organised in a way that is useful in terms of some purpose. By definition, then, a system consists of both structure (order) and purpose (process), and it is this dynamic relationship that lies behind the emergence of patterns.

> A pattern emerges between two entities if it is present in the combination of the two entities, but not in either of the entities separately. And the structural complexity of an entity is defined as the "total amount" of pattern in it.
>
> (Goertzel 1993)

The pattern view is thus a meta-system built on top of a system. However this meta-system is not an add-on, it derives from the functioning of the system. The patterns are just those factors that keep occurring in the expression of system order in function. The patterns are representations of particular details within the system, and the pattern language reflects the order in the system by means of the relationships between the patterns.

The real power of the pattern concept derives from the fact that the patterns are a fundamental aspect of the system, rather than being a view imposed from the outside. Patterns are *discovered* in a system, not created through analysis. Most systems are simply too complex to understand without breaking them down into their constituent parts. But this means that you are never dealing with the system as a whole. Moreover, the interaction of many parts causes difficulties in linear or reductionist analysis due to the non-linearity or circularity of causation and feedback effects. In situations with complex interactions of these sorts, however, patterns are ubiquitous. For example, in many diverse and otherwise chaotic appearing systems, the same pattern in growth and decline of the *system as a whole*, the logistic or S-shaped curve, occurs. So the dynamics of phenomena as diverse as animal populations, stock market behaviour, the volume of world airline traffic, Mozart's output of music, the construction of Gothic cathedrals in Europe, and so on, all display the same pattern in terms of their development over time (Monod 1974, p. 37). This pattern was found by Cesaré Marchetti and it derives from the fact that many factors interact in 'growth' situations such that where F represents a fraction of the final size of the system the ratio F/(1 - F) pertains.

> Marchetti's work strongly suggests that there is some kind of universal principle governing a large number of human and natural phenomena, and that the form of this principle is the simple logistic rule outlined above. In Marchetti's setting, the logistic rule enables us to find structure in what initially appears to be a more or less random scatter of data.
>
> (Monod 1974, p. 40)

The point illustrated by this logistic principle is that wholeness of a system is expressed in its functioning, and *it is at this level* that the patterns are expressed. The implications of this in terms of understanding the system concerned are profound because what it indicates is that patterns are not simply components or parts of the system, they are expressions of the **nature of the system as a whole**, they speak to system synthesis or productive energy, not just to some structural partioning on the basis of some kind of logical analysis, and what we want to do here is to explore these implications for programming. We feel that the major contribution that Alexander's notion of pattern language makes to programming, and, indeed, any creative endeavour, is the ability to allow one to deal with the elements of a system in **a way that does not destroy the system's existence as a *whole* entity in its own right and on its own terms**.

> The greatest hindrance in the understanding of the living organization lies in the impossibility of accounting for it by the enumeration of

> its properties; it must be understood as a unity. But if the organism is a unity, in what sense are its component properties its parts? The organismic approach does not answer this question, it merely restates it by insisting that there are elements of organization that subordinate each part to the whole and make the organism a unity. The questions '*How does this unity arise?*' and '*To what extent must it be considered a property of the organization of the organism, as opposed to a property emerging from its mode of life?*' remain open. A similar difficulty exists for the understanding of the functional organization of the nervous system, particularly if one considers the higher functions of man. Enumeration of the transfer functions of all nerve cells would leave us with a list, but not with a system capable of abstract thinking, description, and self-description. Such an approach would beg the question, '*How does the living organization give rise to cognition in general and to self-cognition in particular?*'. [Emphases in original]
>
> (Maturana 1970, pp. 5-6)

In Alexander's thinking, process is the key to both patterns and the language they form. So, for example, the features of a garden are there mainly to contribute to a process, life[1]. They are essentially concerned with enhancing the human relationship with the environment, CONNECTION TO THE EARTH in Figure 6.1, so their context is the 'human living environment', they balance human concerns such as the need for peace, tranquillity, recreation in the widest sense, social activity and so on, with the topology, climate and other elements of the built environment of the site. But the design of a garden is another process, that of using those features (living patterns) from the experience of gardens that have shown themselves to contribute to the first process. This second "generative process" is driven by the relationship between the 'living patterns', so the web of relationships form a language.

Therefore, a pattern language is a collection of features of a whole system called patterns, along with a description, usually in graphical form, of how to combine them into meaningful compositions. A pattern language prescribes only those configurations among patterns that contribute to system wholeness; it can be thought of as a way of describing dependencies between patterns. So, in a pattern language diagram, such as the one adapted from Alexander (Alexander 1979, p. 314) shown in Figure 6.1, the arrows show a refinement relationship between patterns. Patterns nearer the bottom of the figure refine patterns above them. We say that GARDEN GROWING WILD is smaller than HALF-HIDDEN GARDEN. Intuitively, this means that GARDEN GROWING WILD is applied as a

[1]It should not be forgotten that gardens were originally important contributors to life at the fundamental level of providing food, and the "Garden of Eden" story is a reflection of this former centrality in human affairs, and, indeed, a reminder of the continuing significance of our relationship to the ground on which we live. Everything we eat derives, to some extent, from the productivity of the mineral composition of soil, even fish.

structural refinement inside the structure created by the pattern Half-Hidden Garden, that is, the garden as a whole. The essential *fact* about a pattern language is that all the patterns relate back to the topmost pattern in a refinement relationship. It is this *fact* that makes a collection of patterns a language, and, indeed, makes the patterns patterns rather than simple components. (Porter, Coplien & Winn 2005, p. 233).

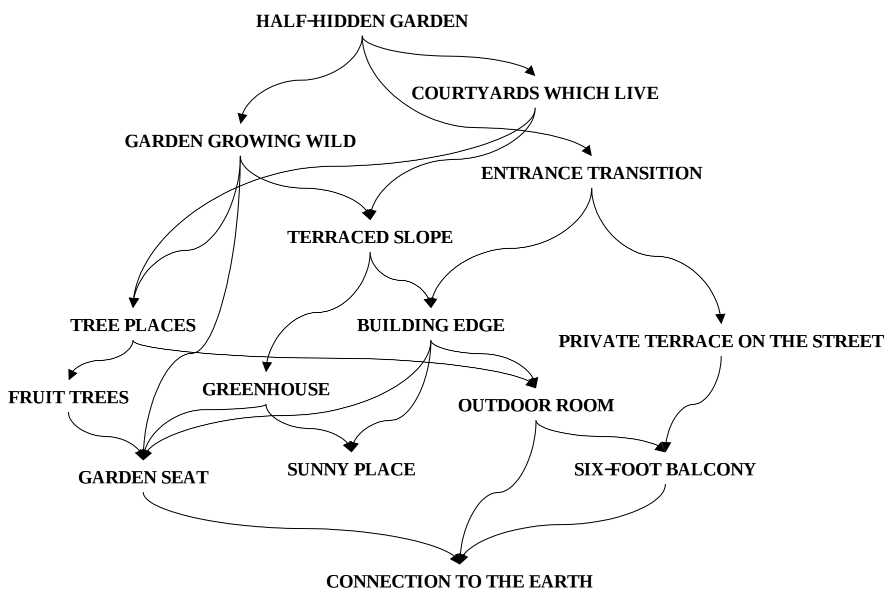


Figure 6.1. A Language for a Garden. Adapted from (Alexander 1979, p. 314)

The important point here is that the very existence of those elements that Alexander calls patterns, and the structure (pattern language) that they form, are contingent on the system functioning as a whole. The patterns are only 'discovered' in the system as it functions, that is, as it is expressing its 'wholeness'. If the system is being conceptually 'decomposed' by the pattern process in any sense at all, then the decomposition is being done on the basis of system dynamics rather than some engineering or knowledge principles that exist independently of the system. We would argue, in fact, that this is not really a conceptual decomposition process at all, because it is being driven by system process, not a conceptual process. It is in this sense that Alexander uses the terms 'life' and 'living'. Like a biological system, a 'living system' is one in which its 'wholeness' is expressed in its functioning, a condition that cannot be guaranteed in a system produced in a mechanical and piecemeal way, as is most often the case in modern architectural design methodologies.

Laymen like to charge sometimes that these designers have sacrificed function for the sake of clarity, because they are out of touch with the practical details of the housewife's world, and preoccupied with

> their own interests. This is a misleading charge. What is true is that designers do often develop one part of a functional program at the expense of another. But they do it because the only way they seem able to organize form clearly is to design under the driving force of some comparatively simple concept. On the other hand, if designers do not aim principally at clear organization, but do try to consider all the requirements equally, we find a kind of anomaly at the other extreme. Take the average developer-built house; it is built with an eye for the market, and in a sense, therefore, fits its context well, even if superficially. But in this case the various demands made on the form are met piecemeal, without any sense of the overall organization the form needs in order to contribute as a whole to the working order of the ensemble.
>
> (Alexander 1964, p. 29)

This would seem to condemn patterns as well, for are not they piecemeal in nature too, one is dealing with a whole system one part (pattern) at a time? The factor that sets patterns apart is that they are not simple products of conceptual decomposition. Consider an aeroplane. Here is a collection of parts that forms a whole. The whole is more than just a simple aggregation of its parts because of its function, flight. No single part, or even incomplete aggregation of some of the parts, will enable this function in a completely safe way, which is why the failure of any part that is involved in keeping a plane in flight will give the flight crew cause to reconsider their flight plan, and to make adjustments as considered necessary to ensure safety.

Therefore, the patterns that are present in modern aircraft are those that are left from the various experimentations around the principles of safe flight and from the normal operations of aircraft over time. That is, the patterns express the functioning of flight from *experience*, not just from conceptual theorising. Parts or subsystems that are found to contribute negatively, or even ambiguously, to safe operation will be eliminated, not through theorising, but through actual experience of function. Patterns are the product of experience, not pure theory. They are present because they have stood the test of function - they contribute to the wholeness of the overall plane-as-a-flying-system not just the plane-as-a-structural-entity. Patterns in programming are like that. They are not just structural elements of the programming system but patterns in the expression of the function of the system being produced.

It is the nature of the problems of explicit design to be different from everyday problems to the extent that the patterns of everyday experience may not be the most appropriate because the design space is not as well understood, one's experience of it is not as deep as the general experience of life. Of course, some "design spaces" are less complex than everyday life to the extent that they can be dealt with mechanically or logically, and these enable us to discover what it is that makes complex systems difficult for designers, what distinguishes design from logic, in effect.

> There are certain kinds of problems, like some of those that occur in economics, checkers, logic, or administration, which can be clarified and solved mechanically. They can be solved mechanically, because they are well enough understood for us to turn them into selection problems.
> To solve a problem by selection, two things are necessary.
>
> 1. It must be possible to generate a wide enough range of possible alternative solutions symbolically.
> 2. It must be possible to express all the criteria for solution in terms of the same symbolism.
>
> Whenever these two conditions are met, we may compare symbolically generated alternatives with one another by testing them against the criteria, until we find one which is satisfactory, or the one which is the best. It is at once obvious that wherever this kind of process is possible, we do not need to "design" a solution. Indeed, we might almost claim that a problem only calls for design (in the widest sense of that word) when selection cannot be used to solve it. Whether we accept this or not, the converse anyway is true. Those problems of creating form that are traditionally called "design problems" all demand invention.
> Let us see why this is so. First of all, for physical forms, we know no general symbolic way of generating new alternatives - or rather, those alternatives which we can generate by varying the existing types do not exhibit the radically new organization that solutions to new design problems demand. These can only be created by invention. Second, what is perhaps more important, we do not know how to express the criteria for success in terms of any symbolic description of a form. In other words, given a new design, there is often no mechanical way of telling, purely from the drawings which describe it, whether or not it meets its requirements. Either we must put the real thing in the actual world, and see whether it works or not, or we must use our imagination and experience of the world to predict from the drawings whether it will work or not. But there is no general symbolic connection between the requirements and the form's description which provide criteria; and so there is no way of testing the form symbolically. Third, even if these first two objections could be overcome somehow, there is a much more conclusive difficulty. This is the same difficulty, precisely, that we come across in trying to construct scientific hypotheses from a given body of data. The data alone are not enough to define a hypothesis; the construction of hypotheses demands the further introduction of principles like simplicity (Occam's

> razor), non-arbitrariness, and clear organization. The construction of form, too, requires these principles. There is at present no prospect of introducing these principles mechanically, either into science or into design. Again, they require invention.
> It is therefore not possible to replace the actions of a trained designer by mechanically computed decisions. Yet at the same time the individual designer's inventive capacity is too limited for him to solve design problems successfully entirely by himself. If theory cannot be expected to invent form, how is it likely to be useful to a designer?
> (Alexander 1964, pp. 73–75)

Alexander's answer to this question is, ultimately, "that there is a deep and important underlying structural correspondence between the pattern of a problem and the process of designing a physical form which answers that problem" (Alexander 1964, p. 132) because shape, be it physical or conceptual, is the expression of some hierarchy of order in function. The human form, in conceptual terms, derives its ultimate power from the transmission of experience through the use of abstract symbolic form. No newborn infant has to develop a pattern language for life, she is, in effect, born into it - we call it culture. What is powerful about the pattern idea, then, is not that the patterns are the same in every situation, that there is one simple pattern language for the whole of life, but that there will be, in every domain, features that are patterns, that function in the same manner that patterns function in everyday living. What crosses over from everyday life is not the patterns themselves, but the way that patterns function in ordering our lives, in 'designing' our living style, our 'life as a whole' in effect.

But there was an important issue that Alexander mentioned in the previous quote which cannot be allowed to pass unnoticed, and that is the "measurement problem" which we discuss in Section 8.4. As Alexander points out, "we do not know how to express the criteria for success in terms of any symbolic description of a form" because the degree of success is a function of process not structure, it is a form of "meaning" not a quantifiable property. The 'measurement problem' arises from the fundamental difference between discrete entities (elemental form) and the continuous and wholistic, not to say subjective, nature of experience (experiential form). Some aspects of higher level form are impossible to represent at lower levels, of organisation because they are emergent properties. Representing a musical piece in terms of air pressure, for example, will not give one a sense of the piece in musical terms (Einstein, quoted in (Born) 1969), there is an epistemic disjunction involved, a "loss" of information (or, more properly, a loss of "meaning"), in the translation. Meaning is a matter of patterns, of prior experience, think of the television drama that is a notable experience for the mother but which remains "for the infant 'an undifferentiated homogeneity' of flickering lights and persistent noises" (Britton 1970, p. 17).

6.2 Dealing with System Complexity

Ultimately, everything that we do is a product of input from the world. Input from the world enters our mind through our senses, our perceptual apparatus. But this apparatus does not function as a simple data collection mechanism as, say, a video camera connected to a computer system does. The process that we call perception is much more sophisticated than a simple input of data to an information processing device. What enters is not raw data as such, but *meanings*. In effect, the data has been 'preprocessed' at the perceptual level. That is, "perception might be conceived of as a set of preliminary 'data-reduction' operations, whereby sensory information is described, or encoded, in a form more economical than that in which it impinges on the receptors" (Attneave 1959, p. 82). What is occurring is that the human understanding system is fitting the enormous amount of data that impinges on it into a form that relates to its current needs and interests so as to best utilise its information handling system. It does this by generating *meaning* from the raw data, by applying a filter to it as it enters. In other words the data enters as *patterns of significance*, meanings in short. The only way that patterns of perception can be generated in this way is through the use of a system that identifies patterns in the raw data, and combines small units into larger ones through a hierarchy of relationships.

The pattern language idea is based on the patterns of recurring activity that occur in systems. Any system is about two main things, order and process - structure and function. Much of the process in a system is based on maintaining and creating the order that lies behind system structure. Any system, natural or artificial, is made up of a lot of detail. Assimilating large masses of detail is not something that humans do easily, rather we understand things by relating them. One way of doing this is categorisation - putting similar concepts into a category. Another way is analogy. We come to 'understand' something that we previously did not understand by relating it to a different concept that we already do understand. And a third way is the sensing of pattern hierarchy, the pattern language, within the larger categories. This is the phenomenon that Gregory Bateson calls metapattern, derived from the fact that, in Alexander's terms, a simpler pattern 'refines', or 'adds structure to' the more complex pattern to which it is related, thereby building, or 'creating', the larger 'structure' that we call *understanding*.

> What is meant intuitively by a "pattern" is, essentially, a representation as something simpler. When one represents x in terms of something simpler than x, one has obtained a pattern in x. A representation is a kind of relation, between the representer and the thing represented. So, as one would expect, the Metapattern reduces self-organization to certain relations between entities. It contends that everything is made of relationship.
>
> (Goertzel 1993, p. 2)

These three ways of handling detail involve patterns in the sense of 'recurring form'. Consider, for example, what is actually happening when you see a horse.

The outside appearance of a particular horse is made up of a large, but finite, number of molecules that define the detail of its appearance as the particular horse that it is. However within this mass of detail occur points of change that show up in all horses. These recurring forms in the mass of detail are the 'patterns' that enable us to match what we are seeing with the image of 'horse' that we possess from previous experience. In any particular horse, then, there is just continuous detail. I can only discern the patterns that tell me that this is a horse because they are common to all horses. The patterns transform the mass of detail into 'horse', separate the particular section of the overall scene into the horse that I see within it. Patterns then are the fundamental basis of category or 'form'. And 'understanding' derives from form not detail. I don't have to apprehend every single detail of the individual in recognizing the form, 'horse'. All I need is to apprehend the patterns that make up the form 'horse'.

Of course, there are other 'patterns' of detail that show up in only one particular horse, or in only some particular horses and not all horses. These patterns of detail form individual or group characteristics. They 'define' at those levels of order rather than at the fundamental level of the form 'horse'. These are things like 'palomino' and 'pinto', smaller categories within the larger one. Note that we keep referring to the fundamentals of 'understanding' here. Categories are critical to reason, they differentiate detail, provide structure, and are the source of order. They form the basis of 'understanding. Undifferentiated, unstructured detail is impossible to understand. We could only begin to understand atomic theory, for example, once we could discern the regularities, the pattern of form that is expressed in the periodic table by the atomic level detail. Understanding comes from the patterns, the recurrences, in detail, rather than the details themselves. So a horse is a mass of detail, but at some point the mass nature of detail 'breaks', giving us a framework on which to base our understanding of 'horse', of 'pinto', and even of features that are unique. We recognize a particular horse by the unique features that do not recur in other horses. So these features are not 'patterns' in the sense that they recur in reality, but they are 'patterns' nevertheless, the recurrence being in our memory of that horse.

We are using the similarities between concepts (commonality) to increase order. Categorising things based on the similarities and the differences between them allows us to 'see' the patterns in the system. We have differentiated the mass; added coherence to the detail. However the need for this view of a system, this understanding, arises from process. The necessity to have reliable processes of identification and the like leads us to discover the patterns.

Let's go back to the horse example for a minute. It is important to see that the patterns have not only differentiated detail, they have done it in a way that forms a hierarchy that carries meaning. In other words the 'pinto' pattern is a refinement of the 'horse' pattern, it tells me that a horse is a particular type of horse. And my knowledge of individual characteristics enables me to distinguish between different individual horses. So, in any scene, the mass of detail is made coherent by my ability to recognize the patterns within it because my knowledge is based on a hierarchy of patterns, a pattern language. This pattern language empowers those processes like identification and recognition that underlie knowledge, that give meaning to the world. And although the patterns in Figure 6.2 are named, this is not necessary in terms of using a pattern language in the head. For example, in this case I might not be aware that the dogs called "Bluey", "Fido" and "George" are members of the same breed, in this case "Blue Heeler", but I will have placed them in the same category on the basis of their appearance nevertheless. In effect the pattern called "heeler" would be "unknown breed", but it would still connect to the other patterns in the same way.

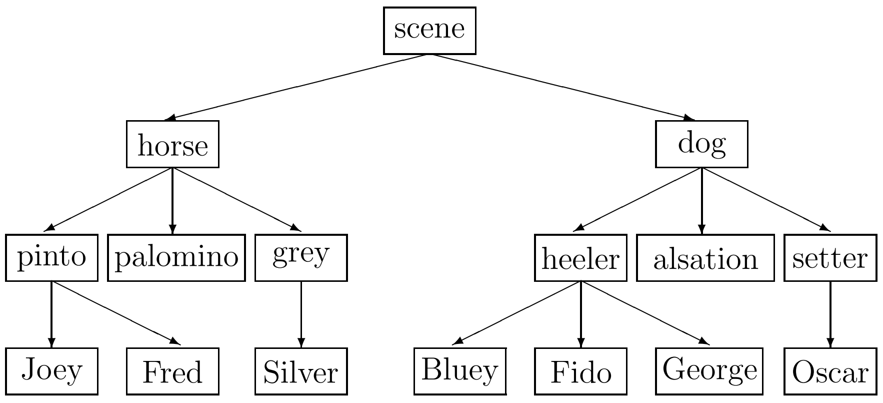


Figure 6.2. The patterns within a scene differentiate the raw detail.

Everything that we do as humans involves using a pattern language. We make sense of a situation on the basis of previous experience. In a classic study, three people, a chess master, a good player, and a novice were given 5 seconds to view a board from the middle of a chess game. After 5 seconds the board was covered, and each participant was asked to reconstruct the board position on another board. The master correctly placed many more pieces than the good player, who in turn placed more than the novice: 16, 8, and 4, respectively. However, these results depended on the chess pieces being arranged in configurations that conformed to meaningful games of chess - in other words the patterns carried *meaning*. If the chess pieces were arranged in random order, that is the 'patterns' had no meaning in terms of chess, all three participants correctly placed from 2 to 3 positions only (See Figure 6.3). This shows that expertise in a domain is based on knowledge of the *patterns of meaning*, in other words, "understanding" that exist within it. The master is an expert because she 'has' the pattern language that enables her to 'play' chess to a very high level.

This relationship between patterns and process is the critical one in terms of programming. A large part of the 'order' that exists in a programming system

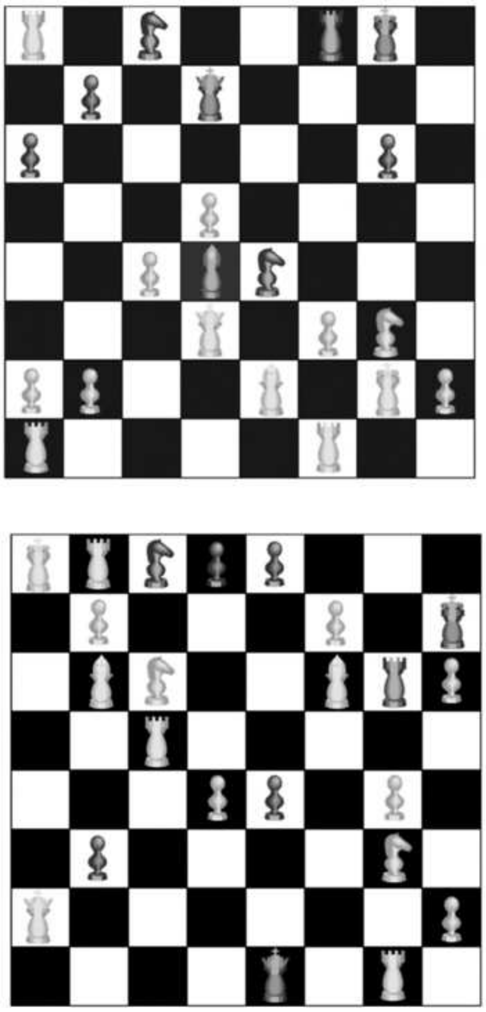


Figure 6.3: Patterns in Chess carry meaning for the expert that they don't for the novice. (Figure from (Bransford) 2000)

is derived from the syntactic structure of the programming language, or, at a deeper level, the way that the syntax sets up the patterns of electron flow. A program is an artefact of the programming environment, therefore it necessarily reflects the 'order' of this system, the syntactic structure of the language. Solving any Java programming problem, for example, will involve applying the elements of Java syntax, so the process of design is, at least in part, syntax driven. The ramifications of this are that when a system of any kind is computerised, the programming language system used becomes the means of development for that system. You can only interact with system order, in the sense of changing or maintaining it, through the programming language system from that point on.

So what you have now is a real-world system that includes a programming language system. Any holistic view, therefore, must take account of that fact. Whatever the system is in terms of the service it provides is now conditioned by the fact that it runs on a computer. So what you are really doing is adding a programming tool as a means of addressing the order and functioning of the system. You are deciding that it is more efficient to deal with problems in this system using computers. Ipso facto you are now going to have to view all the problems in those terms. The patterns and the pattern language will now be those of the programming system.

6.3 Coherence and Balance

At this point in documenting a pattern we hit a major difficulty, an epistemic disjunction, in fact, because the person who is documenting the pattern can only, in some sense, guess at what was in the mind of those who originally established the element of practice that is being documented as a pattern. What a pattern author sees is the actual artefact as it recurs in her experience of the world, not the circumstances, *the whole context*, behind its production. All that one can know about a pattern from its existence as a pattern is that for it to have become a factor that recurs it must be, in some manner, balancing the forces that led to it coming into being. We have a similar difficulty in defining evolutionary fitness - saying that a species is 'fit' because it survives is almost circular in terms of our definition of evolution as the survival of the fittest and a species continued existence can tell us little about the complex of factors involved in its continuing survival.

This is also one of the points where we touch on the element of moral coherence that Alexander places great emphasis on and which is mostly avoided by pattern practitioners precisely because morality, for the modern mind, exists at the same intersection of the subjective and the objective. We come back to the moral order question later (see Section 11.4), but at this point it is the practical consequences for a pattern author of the subjective-objective gap that interests us. The artefact that is causing the pattern to be written down is a factor in the objective world, but the thinking that produced it lies in a subjective world about which we can only surmise. This is why, to Alexander, the notion of patterns existing as elements in a language is so crucial. Achieving a good fit between form and context is the *very purpose* of design, and therefore can be assumed to be a factor in the thinking of the designer. The fact that the pattern is a pattern, that is, that it recurs over time, says that the purpose of the designer, a good fit between form and context, has been achieved. But in order to have accomplished this feat the designer must have had what Alexander terms a "field description" of the problem domain.

> Let us suppose that we did try to write down a list of all possible relations between a form and its context which were required by good fit. (Such a list would in fact be just the list of requirements which designers often do try to write down.) In theory, we could then use each requirement on the list as an independent criterion, and accept a form as well fitting only if it satisfied all these criteria simultaneously.

> However, thought of in this way, such a list of requirements is potentially endless, and still really needs a "field" description to tie it together. Think, for instance, of trying to specify all the properties a button had to have in order to match another. Apart from the kinds of thing we have already mentioned, size, color, number of holes, and so on, we should also have to specify its specific gravity, its electrostatic charge, its viscosity, its rigidity, the fact that it should be round, that it should not be made of paper, etc., etc. In other words, we should not only have to specify the qualities which distinguish it from all other buttons, but we should also have to specify all the characteristics which actually made it a button at all.
> Unfortunately, the list of distinguishable characteristics we can write down for the button is infinite. It remains infinite for all practical purposes until we discover a field description of the button. Without the field description of the button, there is no way of reducing the list of required attributes to finite terms. We are therefore forced to economize when we try to specify the nature of a matching button, because we can only grasp a finite list (and rather a short one at that).
> (Alexander 1964, pp. 24-5)

At the time that he wrote this, the pattern language idea had not yet firmed in Alexander's thinking. Nevertheless, it is clear from the example used in the quotation that what he is referring to here as a "field description" is, in fact, a pattern language. The way around the impossibility of handling the infinite list of possible criteria in the matching process, is to *set the context* for the matching process to ever narrower terms, just as a pattern language for keying biological specimens does (see Section 10.4). Of course, in a simple matching process, there is a 'natural' congruence between force and context, called here a "requirement" or a "criterion" for the match which *does not* occur in the more complicated process of design, and to that extent the example is a bad one. Nevertheless, even though the forces cannot be dealt with one at a time as they can be in a matching process - eliminate all buttons of the wrong colour, eliminate all buttons with the wrong number of holes, etc. - the principle is the same, a pattern language is refining the design by setting the context to an ever finer level of detail. The form, or shape, that a particular pattern language displays, is a reflection of the "field description", the contextual structure, of the task at hand. "You need to have some global awareness of how the thing has got to be laid out before you start attempting to go into detail. And that of course the pattern language [has] to deal with" (Alexander quoted in (Grabow) 1983, pp. 94-5).

In the foregoing, we have skipped over an important concept that seems to be widely misunderstood in pattern practice. In the statement that design is the act of fitting form to context, the term 'context' is being used in a particular way. Form is dealing with that part of the environment, seen as a whole, that is being manipulated by the designer, in other words, the elements of environment that we can directly control. So 'context' cannot simply be synonymous with the term 'environment' as it normally is, because form is as much a part of the environment as context is. Context, in this scheme, is *that part of the environment which puts demands on the form* (Grabow 1983, p. 36). That is, it is *the full constellation of forces*, acting holistically, that the form needs to balance in order to achieve a good fit. But the critical point here is that the balance is achieved by the *whole* artefact ("form and context are in frictionless coexistence" (Grabow 1983, p. 36)), not by the individual components - it's the plane that flies, not the parts of which it is made and this is why it is *the pattern language* that has to balance the forces, fit form to context, *not* the individual patterns. In fact, each pattern, as it is applied, can only deal with the context as it exists at the time of application in the whole design process, that is, the context that was set up by the previous pattern, not the *context as a whole* that the form has to fit.

So context is , in fact, nothing more than the *total effect of all the forces* on the form, and the pattern language, the diagram, puts as much of the original contextual structure back into the design process as is likely possible. This is the main reason why Alexander insists that "the real work of any process of design lies in this task of making up the language, from which you can later generate the one particular design. You must make the language first, because it is the structure and the content of the language which determine the design" (Alexander 1979, p. 324). It is the relationships between the patterns in the language that provides most of the information required to unlock the generative potential involved in the conceptual structure of the objective domain. What Alexander says about this is that it is the objective contextual backbone of the domain, "*the structure of the network* which makes sense of [the] individual patterns because it anchors them and helps make them complete [emphasis added]" (Alexander 1979, p. 315). This notion of patterns being anchored in and made complete by their placement in the pattern language is important in another sense, the tendency of the modern mind to make a radical separation of patterns from the past from their historical context. It is exactly this tendency that leads to many of the confusions and uncertainties that we see all around us because it posits a sort of individual absolutism - beauty lies in the eyes of the beholder - or, at best, a form of cultural relativism - slavery was morally correct in earlier times because most people agreed that it was.

The trouble with this separating tendency is the complete loss of context involved. Patterns like aesthetic appreciation and slavery were anchored in the pattern language of the time, and did not *necessarily* speak to any objective notion of beauty or moral order. They derived more from the social structure, the particular groups that controlled the formation of the prevailing cultural norms, for example, than from genuine community feeling. If I have a major criticism of Alexander, it is in regard to his idea that the Quality Without a Name springs *automatically* from community. Community is not an absolute good. It can be controlled and manipulated, one has only to think back to the

Nazi period in Europe, so the idea that a pattern language automatically reflects real underlying community values, let alone any absolute human values, has to be treated cautiously. But even in the case where a pattern is separated from a pattern language that is assessed to be free from corrupt influence, the pattern becomes incomplete, essentially meaningless, and even dangerous. This is why the "making up the language" is so critical. It is only safe to use patterns from the past if their context is properly readjusted to the modern situation because their position in the pattern language of their time does *not* guarantee their appropriateness in the *new context* - the *new* demands that the environment is making on form.

There are many examples of the danger involved of lifting patterns out of their historic context in this way but a particularly disastrous case has become apparent on the Indian sub-continent in recent times. In response to the deteriorating quality of surface water in Bangladesh and other parts of the Indian subcontinent due to biological pollution, overseas aid and local government agencies drilled hundreds of thousands of tube wells to tap biologically uncontaminated underground sources. This *pattern* had, of course, been used for millennia in drier parts of the world where surface supplies are few, or even non-existent. However, applying it in India and Bangladesh involves lifting it from a context where historical practice had shown it to deliver water suitable for human consumption. Given the *completely different context*, the *novel configuration of the forces* involved, the pattern should have been applied with extreme caution, that is, it should have been embedded in a pattern language developed specifically for the new situation rather than being applied in isolation. The mere fact of changing the context meant that the balance of forces achieved by the historical usage could no longer be guaranteed. Unfortunately, nobody undertook the task of "making up the pattern language" for the new context and the biologically unpolluted underground water turned out to be contaminated with a geo-chemical pollutant, arsenic, chronically poisoning millions of people.

After all, the main thrust of Alexander's thinking is that *only* by the use of a pattern language can *morally coherent* outcomes be produced. Patterns applied in isolation cannot do this, and I would say that this is precisely because a single pattern cannot list all the possible 'forces' at play in the situation, that some of them need to have been 'eliminated', in effect, by the application of prior patterns in the language, and, indeed, some of them will become apparent only *after* the application of the pattern currently under consideration. The moral force of pattern language, its expression of wholeness, comes from the fact that it puts the problem in the wider context in a way that the single pattern never can. It deals with *all* the forces involved by putting every pattern in its correct context in the language. Indeed, what I would say about context and forces is that these are issues that derive from the *problem that is being dealt with*, they are *not* just a feature of a pattern. In the India/Bangladesh case the problem, supplying safe water to a community, exits in a *different context* from the historical precedents, and the pattern language used has to deal with *the problem in its context* exactly as Alexander says.

It seems to me that the whole point of the pattern language diagram is that it is based on *context*, the total effect of all forces, a concept much more general in scope than individual forces, and this is what gives it its power. In some sense it allows the user to ignore the details, *including any forces*, in the actual situation, at least temporarily, while the conceptual outline of a solution, its design, is constructed. The whole thrust of this dissertation is that there is a fundamental difference between an entity or a process as it occurs in the 'real world' and any representation in the abstract, in that the representation is formal or 'ideal' - 'ideal' in the sense that it exists as a set of *ideas*, not that it is perfect in the modern sense of 'ideal'. As an example take a university PhD program. In one sense, this 'exists' as a formal process, an 'idealised' version if you like. But this version is abstract in a way that the actual process as it works through in practice in any particular PhD project simply cannot be, and this is what makes it formal or 'ideal' rather than 'real'. So, for example, in the official policy sense, the formal version, it can be stipulated that the candidate should not be aware of the identity of the examiners of her thesis. But, in 'real' terms, as a formal requirement, this founders on the fact of the particular relationship between a particular candidate and her particular supervisor. Their personal natures are 'forces' in the actual candidate-supervisor relationship as it exists in reality in a way that cannot be encompassed in the formal representation of the process. The academic 'ivory towers' that concoct and administer the 'formal' PhD process necessarily have a representation of the candidate-supervisor relationship that is 'ideal' in nature, not 'real', and therefore they simply can't be cognisant of all possible forces in terms of an actual human relationship on the ground.

The central thesis of our dissertation is that any 'real world' process has to deal with *all* the forces that pertain in the actual situation on the ground, so to speak, but that *no* representation in any formal system can do this, it is simply impossible, which is why the programming language (machine logic) is *not* sufficient means in learning to program. In terms of this project, then, it follows that the pattern writer, insofar as she is formalising a real world reoccurrence, cannot anticipate all the factors that will ever be involved in situations where her pattern will be applied. Patterns might be identified by their recurrence in the 'real world', but in documenting them one is converting real world entities into conceptual ('ideal') form and this *always* involves abstraction, that is, a loss of some of the information inherent in the actual occurrences that led to the identification of the pattern. One is necessarily taking them out of their original context in the real world and this loss of context is dealt with, in Alexander's scheme, by embedding them in a pattern language being made up for the new context.

So, in our field, what is the pattern practitioner to do? Unfortunately, in adopting Alexander's ideas we have largely ignored the language half of the 'pattern language' formulation. The consequence that flows from this is that we have had to deal with forces directly in the pattern form in a way that is *not done* in Alexander's patterns[2]. His method is to rely on the forces being balanced by the whole of the generative process that derives from working through the pattern language formulated for the particular situation. In a sense, the pattern language is dealing with the forces automatically, although one has to appreciate that this is a 'natural' rather than a mechanical automaticity (it derives from the context, the non-controllable part of the environment in which the pattern is being applied - see Section 11.4), so that the pattern author (formaliser) does not have to explicitly deal with them at the individual pattern level, they are elements of context that can be safely ignored (abstracted away) *by the pattern author*[3]. However, given the lack of language in pattern practice in the software field, the pattern user, as the agent who has to "make up the pattern language" applicable to her particular context, is therefore reliant on the pattern author correctly identifying *all* the forces involved because the pattern author has not supplied the contextual information provided by the relationships between the patterns.

By abrogating the task of making up the language for the context in which the patterns were discovered, the pattern author is implicitly depending upon the pattern user to make up the language for the user's context without any guidance from the patterns' original context. In a highly technical field like programming, and given users who are experienced in doing this, it works. But, as the water supply example demonstrates, it is only safe to do this if the pattern author can rely on her pattern being applied by *a community of pattern practitioners who are experienced in making up a pattern language from scratch.* As pattern usage in programming has been conducted in the main by experienced programmers, pattern authors have been able to get away without pattern languages because they can rely on the users having a pattern language for programming, at least of sorts, from their general programming experience. However, it is the very point of our argument that programming novices cannot be expected to have this acquired-by-default pattern language for programming. Indeed, it is the instructor's job to *facilitate the development of same* in the learner's mind. So, it is important that the pattern author keeps this salient fact in mind at all times. What the novice needs is a sense of structure, the *context*, or that part of the environment that is placing demands (forces) on the form, and structure is simply not exposed by a linear list of forces presented in the pattern form.

This is not meant to imply that such a list *should not* be given, just that, as

[2]Where he does talk about forces within the pattern form it takes place in the form of a discussion which is more of an exploration of context, a "field description", in his terms (see (Alexander 1964, pp. 24-5)).

[3]It should be noted, of course, that this is decidedly *not true* of the pattern user and it is for this reason that Alexander insists that the user *must* make up the pattern language first because it is only the user who can be aware of the particular context - and therefore the particular configuration of forces pertaining) - to her problem.

matters stand in our field we have come to rely on it almost entirely - but the point is that this is an accident of history, a result of the failure to adopt the pattern language concept fully. Individual patterns, in themselves, can convey no sense of structure, so without the language diagram, the novice is being forced to create it for herself, we are back to relying completely on *the innate creativity of the individual.* All that the list of forces can do is suggest the sort of factors that the pattern user should be looking for, it does not, and cannot, illuminate how the forces interact to provide the context. Again, it is that same confusion that Alexander's ideas were meant to address, the difference between a piecemeal and a holistic approach. Context is *the whole balance of forces* that is currently causing the problem, the balance of forces that is being *reformed* by the design process. Creating new form is nothing more than *reforming the environment as a whole*, and the factor that tells you how to address the problem, how to achieve the needed fit between form and context, can only be the *context*, the particular configuration of forces that needs to be rebalanced.

The balancing act performed by the language half of the equation is probably best illustrated by a common everyday task such as facial recognition. If I asked you to list the features of the face of someone that you know really well (forces), you would, almost certainly, struggle, and, moreover, the description that resulted would, anyway, prove utterly useless in enabling recognition of the person by its readers. Yet present people with a photograph, or even just a realistically rendered drawing of the face, and the situation changes dramatically. So what is it that the likeness does that the list of characteristics fails to do? I would say that the difference is encompassed by the word 'balance'[4]. The likeness does not just present the features (forces) one by one, it *balances* them to produce a representation of the face *as a whole.* It is the *holistic* nature of the visual representation that empowers recognition *not* the individual characteristics of the face as the embodiment of the forces that give rise to the form it takes. Indeed, the *art* of drawing or painting, surely, is the attempt to attain this balance. Even in those cases, such as caricature, where one particular feature is exaggerated beyond all reality, the artist's task is, nevertheless, still to render the balance of the other features in terms of the deliberate distortion of the one such that the whole is still recognizably the subject of the caricature.

So how does recognition work? Is there some list of features stored in the brain against which one 'measures' real faces that come into one's vision? Most certainly not! For all our almost total lack of knowledge of the neuronal processes involved we can be certain that it is *not* a mechanical ticking off of features in a list. If anything is, facial recognition is a classic pattern process, where the features function as a whole, not as individual forces - it's the pattern that matters. Moreover, facial recognition is one of those cases where the pattern (the

[4]For example, listing a characteristic such as 'large nose' is meaningless unless you know something about the rest of the face, the *context* for the nose. A nose is large only in relation to the other features of the face, that is, it is *relative.*

face in this case) can most clearly be seen to be made up of smaller patterns (the particular features are themselves patterns in terms of other faces), that, in fact, form a pattern language in the mind. I would go so far as to say that a system of facial recognition based purely on a straight verbal list of features is impossible. Written systems that do act as recognition aids (in police work, for example) have to be rendered in hierarchic (pattern language) form. So, even here, it is the organization, the *implicate order* in David Bohm's term (Bohm 1980), that arises out of the interaction of the forces that empowers the recognition process, just as it is in the normal visual system. Looking for order in the individual forces is looking in the wrong place.

But there is still a dilemma here because the pattern field as a whole encompasses two separate processes, or three if one persists in regarding pattern languages separately from patterns. Before any process of design can be *explicitly* based on patterns rather than the more usual *implicit* dependence that gets represented as individual creative genius, the patterns have to be identified and documented. So, at this point, the objective-subjective gap has to be breeched in some manner even though it is, in principle, impossible to go back to the state of mind that produced the artefact being documented in pattern form. And here lies the import of the moral dimension in Alexander's thinking because coherence exists at this point in the *wholeness* of the artefact(s) being mined for patterns and can only be propagated into the future from here[5]. The patterns are being documented because the pattern author has appreciated the artefact as a coherent, "morally sound object" (Alexander 1999) in the particular setting in which it exists. That is, the form balances the context. Of course, it is still the case that the balance being apprehended is that which exists at the time that the pattern author is making the assessment, not that which existed at the time of the creation of the artefact. Nevertheless, even though some extrapolation is involved, it can be assumed that the passage of time has not altered the essential objective relationships too much or else the form would have ceased to function holistically.

To me, the situation at this point is quite clear, the pattern of the artefact as a whole, the house-garden, cathedral-ground, program-application, or whatever, can only be rendered in pattern language form as this is the only way that any semblance of the balance can be documented. However, given either that the artefact is documented as a single pattern, or that the sub-patterns within it are presented in isolation, there is no way to represent or describe the balance of forces - remember the pattern language is most often presented diagrammatically. Therefore, the only way that forces can be considered at all is by means of a list, or, at best, an attempt at a verbal description, because context, as *the constellation of demands* put on the form by the environment cannot *easily* be rendered verbally. Of course, Alexander's formulation 'breaks' at this point, the individual pattern can no longer be guaranteed to produce "coherence, morally

[5]The point about moral coherence is elaborated further in Section 11.4.

sound objects" (Alexander 1999). This is now the responsibility of the pattern user, and the list of forces is the only means of giving the user any notion of the factors that will need to be balanced in order to achieve the "coherence" exhibited by the "morally sound object" undergoing documentation.

The ultimate consequence of all this is that if it turns out to be true that it is not possible, because of the highly technical or complex nature of programming, for the pattern language form to function in software development generally, that only individual patterns have a place, then some of the claims that Alexander makes for his formulation are not achievable, and we just have to live with that. However, whatever the case turns out to be in advanced programming, we do not believe that this can be said of the situation in novice programming, and it is the purpose of our project to establish that, indeed, pattern languages are not only possible, but *necessary*, in order to break the impasse caused by the difficulty that novices currently encounter. But the pedagogical sector also has a big advantage in terms of pattern language usage and that is that the knowledge base being represented is in a constant state of flux, such that elements that need to be explicitly depicted at an early stage (`SWAP` is an example) can be dropped as a sub-language of the full pattern language for later stages as students become used to the particular demands that the programming context makes on operations that are familiar from everyday experience (such as swapping things around). Because the full pattern language 'evolves' over time the complexities of balance can be managed pictorially without blowing the diagram out to unmanageable proportions. Furthermore, it means that the presentation of forces within the pattern form can be restricted to those that are *absolutely necessary* to inform a choice between different means of achieving the same end, loop-recursion for repetition, for example, and which cannot be easily rendered pictorially. In the 'psychological field' that everyone brings with them into the classroom, the 'field description' of many operations common to everyday life, such as making choices, swapping, sorting and repetition, already exists. It is the *context* in Alexander's sense, the *extra* demands (Grabow 1983, p. 36) that the *programming environment* puts on these common operations, that causes the difficulty that novices have with them in the programming classroom, the *particular form* that they take here, not the general form they present in everyday life. So the answer can only lie in new form, form that reflects the contextual background of the programming domain.

6.4 A Pattern Language for Novice Programming

The first thing that needs to be noticed about any such pattern language that it is, in essence, ultimately about programming. Therefore it must cover the same ground as the programming language used in instruction. We have chosen Java as our example in this chapter as it is the language used in the first programming course here at Flinders University at this time. So, in a way, what is happening is that, as pattern authors, we are choosing Java as the artefact from which we are 'mining' our patterns and therefore our language will to a large extent reflect the syntax of Java. In fact, what such a pattern language does is to represent the programming language syntax in the structure of the pattern language. For example, in a Java program, a class *must* be defined first, before any other programming statements are used. This fact of Java syntax *means* that in a pattern language for Java programming, the pattern `CLASS` must appear in the diagram immediately below `PROGRAM` and before any other patterns - `CLASS` refines `PROGRAM` and all other patterns refine `CLASS`. So what is a syntactic relationship in Java is represented as a semantic relationship in the pattern language diagram, because, after all, syntax is mostly context, it specifies where a particular construct is allowable, and where it is not allowed. A large part of the need to assess context and forces in the problem domain is therefore obviated - reduced to a choice between the patterns pointed to by the the pattern last applied, and this "balance of forces" has been derived from the artefact in which the patterns were identified, Java itself in this case. All that is left to decide is which of the patterns applies to the part of the problem currently under consideration as illustrated in Chapter 7, that is to match the current state of the solution, the remaining problem, with the patterns that the language diagram indicates are currently available.

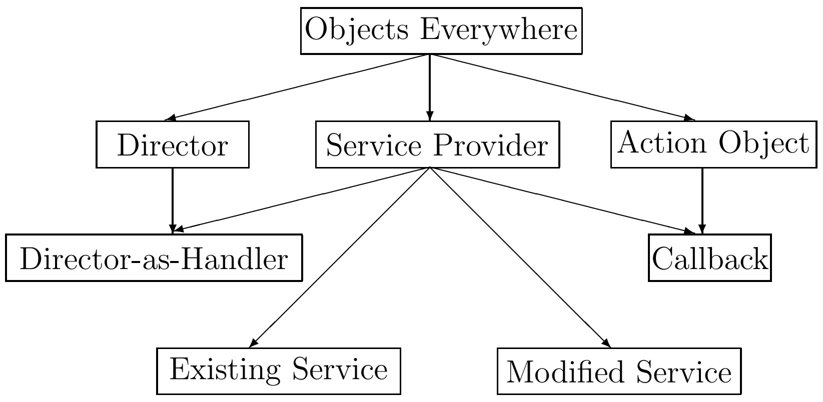


Figure 6.4. The Pattern Language for Object Identification.

So if we start our language with patterns based on those Java constructs that a novice is going to need early on in the course we will end up with a tree that shows the patterns in the context in which they will occur, that is, their programming language syntactic context, modified to some extent by the inclusion of patterns for more general concepts like repetition, making choices, swapping values and the like. These non-Java concepts provide a clearer match with the contextual relationships expressed in the problem space than the patterns based directly on the programming language. But there is another consideration to be taken into account before we get to programming language features, and that is that most programs of more than trivial importance will be made up of more than one class. Accordingly there are grounds for providing a pattern language for designing a program at this level, based on thinking about the objects required. The artefact produced by the use of such a pattern language, an example of which is provided in Figure 6.4, is a class diagram. (Note: Two examples of patterns from this language, and two from that illustrated in Figure 6.6 are included in Appendix A).

However, for our purposes here, this is mostly a diversion, as what we are trying to do is to explain how the pattern process is driven by a pattern language. So the exact details of the pattern language, or languages, used are not significant - the simpler the relationship can be kept, the better. Indeed, in order to avoid complicating the discussion of process in Chapter 7 we have chosen to present it using a pattern language based on the syntax of C rather than Java, because this thesis is ultimately about describing how novices can be provided with the means for generating the sense of process that they mostly lack, not providing an actual 'real world' pedagogical tool.

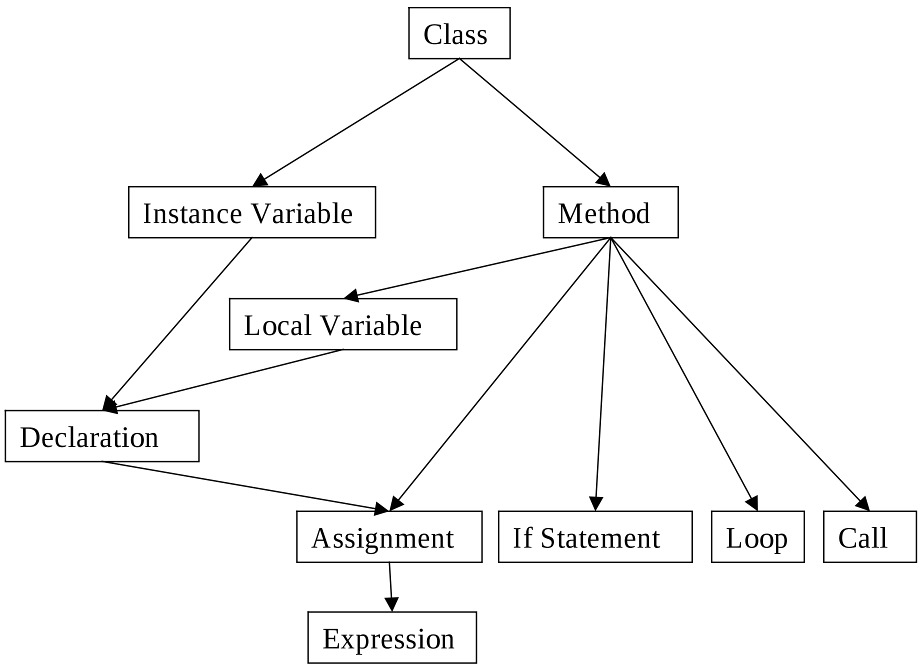


Figure 6.5. Simplified Pattern Language for Class detail.

In Figure 6.5 the arrows mostly represent Java grammar, that is, this is the 'raw' pattern language derived directly from the artefact being mined. So you would start with the topmost pattern which tells you that any Java solution must involve the defining of a class. The arrows from `CLASS` indicate that the next patterns to be considered are `INSTANCE VARIABLE` and `METHOD` and the fact that they are solid says that they are compulsory. This does not mean that any particular class has to have any instance variables but that the pattern must be consulted and the need for such variables assessed. Normally one would assess the need for the class to have any instance variables first, but one can always backtrack through the 'tree' if such a need is discovered later, the position of the pattern in the language diagram reminding one where the declaration will appear in the class definition. If any instance variables are needed the arrow from `CLASS` to `INSTANCE VARIABLE` tells you that they must be declared in the class but outside of any method. The only arrow from `DECLARATION` is optional at this stage of the development process and there are some grounds for indicating the optional status of a link by making it a dashed rather than a solid arrow. As our purpose here is to illustrate the generalising of the pattern language based on a programming system, not to develop an actual pedagogical system, we have kept the diagrams as simple as possible. Having dealt with the need for any instance variables, either in the positive or the negative, you revert back to that point in the diagram where the line of development that led you to `DECLARATION` began, in this case, `CLASS`, and the diagram then suggests that you need to consider `METHOD`. From there the arrows point to the options available when writing a method.

Thus the methodology is to start at the topmost pattern and follow the arrows. So the diagram is a means of setting the context for the next Java construct to be considered. The arrows from a pattern tell you that the patterns they lead to are considered next. Unless two patterns are either directly connected themselves, or connected to the same pattern above themselves, the Java constructs that they represent cannot appear in the same coding context, that is, next to each other in the code. If the connection between them is direct then the levels of each in the pattern language determine in what order the Java constructs can appear, that is the higher before the lower. Being connected to the same pattern means that either that order is not relevant or that the one furthest to the left is considered first, and so on.

However this initial diagram considers only those constructs that occur in Java syntax and is therefore not much more flexible than the raw syntax. Its main benefit is that it does, at least, draw the "cognitive map that we use for solving problems" (Hiltz & Turoff 2005, p. 62), something that is not normally presented to novices. One of the values of the pattern language idea is to reduce the rigidity caused by designing in terms of code. Therefore the first thing to do is to make this initial diagram more flexible in terms of its use by novices by adding concepts, like 'repetition' that are not strictly constructs in the programming language, but are useful in design terms. The point is that the language then becomes a flexible representation of Java syntax directed specifically at the design process for those who do not yet have a developed 'feel' for it. The flexibility comes about because we can make explicit those relationships that are implicit in Java itself. We can define "patterns which specify connections between patterns" (Alexander 1979, p. 187). For example, we can add the pattern called 'repetition that has no direct counterpart in Java, but is implicit in the sense of being a meta-concept. That is, both of the Java constructs, loop and methods, can be seen as means of repeating actions. In the case of methods this might be slightly obscure because the primary purpose of methods is to define a group of actions not to repeat them. Nevertheless a method can be used for the purpose of defining a repeating activity. In this case it is fulfilling the role that the construct, subprogram, fulfils in other languages, so we can use that name for the pattern here, even though in Java terms it will be realised as a method.

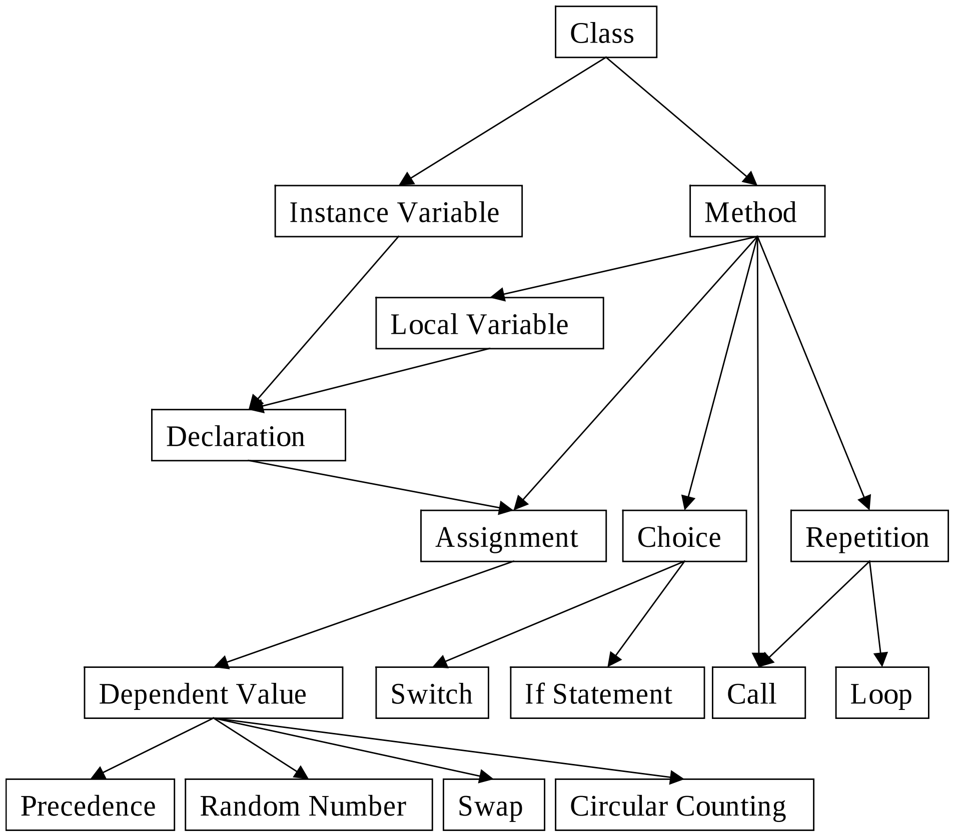


Figure 6.6. Adding Flexibility to the basic Class Detail Language.

This flexibility is important in terms of both the pattern language idea, and the use of this particular language for teaching programming. A pattern language is about designing solutions not implementing them. It can, and should, provide guidance in terms of both design and implementation. The addition of a meta-concept pattern like ‘repetition’ can thus be seen as adding a design principle to the basic Java syntax. You are designing for repetition, even though in Java itself you cannot implement it directly. The connections below the ‘repetition’ pattern direct you to the implementation details. This is what Alexander means when he says that “each pattern is itself a pattern of smaller patterns” (Alexander 1979, p. 185). In our case the pattern language representation of Java can identify the identity pattern "repetition" between the two Java constructs even though Java does not do this itself. A pattern language based purely on the syntax of Java would hardly be more flexible in design terms than the programming language itself - you might as well just implement the solution without worrying about design.

What has been identified in considering repetition as a meta-concept of the programming language is the true context of loop and subprogram in terms of the problem. Seen purely in terms of Java syntax the context of loop is a block, and the context of a subprogram is a class, but these factors tell us nothing about the structure of the problem domain, just the conceptual structure of Java. Adding a pattern for repetition is thereby a way of representing the problem context for the subsidiary patterns in the model of the programming language that is being used for design - extending the basic language to include concepts that speak to the problem specification. Of course, this could be said another way. Identifying a force for repetition in the problem specification indicates that the following patterns are applicable.

We have a discussion here about the precision with which context and forces should be specified in patterns. I think it is important to identify the issues on which this discussion turns. If preciseness is required for pedagogical reasons then it indicates that we probably need a *different* pattern language. The point about the pattern language is that it provides, in itself, the contextual details. It tells me that, at this point in the design process, I need to look for the concepts pointed to by these arrows in the problem specification. Thus, for example, if I wish to more precisely define the patterns available in the `STATEMENT SEQUENCE` pattern, then I need to specify that in the language, not the pattern.

As an example, consider calculation. Calculations often involve repetition, but this might not be obvious to a novice. I can make it more explicit by adding a pattern called `CALCULATION` to those pointed to by `STATEMENT SEQUENCE`. `CALCULATION` would then point to `REPETITION`. This is a much clearer representation of the context than trying to specify it in the `REPETITION` pattern itself. It is the pattern language that should be encoding the *relationships between the patterns*, that is, the *context* of each, not the pattern form. An arrow in a diagram is a much clearer representation of context than any verbal construct. It is why we have maps rather than descriptions for the location of physical objects in space. Let the pattern language provide the pointing, the context, because that is what it does best. Unless there are good reasons, in pattern terms, for trying to precisely specify context in the pattern form, this job should be done by the pattern language.

Another factor here is that information about context is itself contextual. In talking about a context you are going to have to refer to other factors in the total context - "Australia is in the Asia-Pacific region" does not make much sense as a description of Australia's geographical context unless you already know the context of the latter. A map, or a language diagram, avoids the need for contextually recursive verbal descriptions, and should be preferred unless there are compelling reasons for the information to be in the pattern. In purely pattern terms the purpose of the contextual information is clearly to aid in pattern selection. If the pattern language is about the relationships between patterns then it, not the patterns, is the right place for contextual information. It is difficult to see how placing it inside the patterns is helpful in terms of using them.

Yet another factor is the ease of assimilation. An important aspect of pattern theory is the notion of internalisation of the patterns. You are an expert when you no longer have to constantly refer to the pattern documentation because the information is encoded in your brain. The cognitive process is itself heavily based on patterns. Information that can be fitted into an existing structural form is much easier to assimilate than isolated 'facts'. That is, all the information in our brain is organised in terms of the relationships between the individual ideas. It forms pattern languages in fact. Thus having the information about programming organised in pattern language form is clearly advantageous to the assimilation process. It comes pre-digested in some sense - organised in the way that the brain organises information itself, that is, in contextual form.

6.5 Context - problem or pattern?

There is still a decision about which of the two possible meanings of the word 'context' is appropriate here. Which 'context' is being addressed in any component with this name in the pattern form - the context of the problem or the context of the pattern? This is probably not a real question at all because it is difficult to see how these two can be different except in very fine detail. The context in which the current state of the problem exists is created by the application of the previous pattern - the particular problem now being dealt with is the result of the interaction between the pattern language and the original specification, as illustrated in Chapter 7. Patterns can only generate solutions by generating new problems - subproblems in effect. If the application of a pattern does not generate a new problem (subproblem) to be solved then the original specification has obviously been met. Thus the context of a particular problem is in large part created by the previously applied pattern having resolved some of the forces and thereby set up the new context - the intermediate configuration of forces that now needs to be dealt with. This suggests that the context of the pattern in the pattern language is the most pertinent aspect in the process of generating a solution.

Ultimately, this is the justification for concentrating on context. Trying to analyse the forces in a given situation *at the pattern writing stage* is both difficult and dangerous, because the forces exist in the problem space and it is not always the case that the artefact(s) inspiring the pattern author contains a resolution of all the forces in the user's problem space, as in the water supply case outlined above. But Alexander's idea is that the patterns provide the resolution of the forces in the problem space and exist, therefore, in a different conceptual space, that elucidated by the Pattern Language that is "made up" by the user.

> Alexander's descriptions of patterns includes the idea that a pattern should represent a kind of equilibrium of forces. ... This is the same notion as optimality as seen for example in the analysis of algorithms in computer science, but applied to the kinds of harder-to-measure forces described in the previous question. It is usually impossible to analytically "prove" that a solution optimally resolves forces. (In fact, it is hard to define the notion of "proof" here, or even to see what use such a proof would have.)
>
> (Appleton 2000*b*)

The very point about the Pattern Language idea is to shift the task of solving a problem from the conceptual domain in which it actually exists because solving it there *requires* detailed analysis of the whole domain and this is extremely difficult *without a "field description"*. The pattern language provides the field description of the actual problem domain by means of the correspondence between the context of each pattern in the language (the relationships between the patterns) with the context of the problem in Alexander's sense, the full configuration of all the forces.

The difference between the two conceptual domains here, the 'problem space' and the 'pattern space', is analogous to that between an actual physical space and a map of it. The problems that exist for a person in terms of location in a physical space are caused by the inability to see, from within the space, the relationships between the topographical features that exist in it. A map of the physical space replaces the need to make a detailed analysis, a thorough exploration of the whole space, in other words, by resolving the forces - representing the relationships between the features, the *context* in fact. In a vital way, having the contextual relationships of the features of the whole space set out in an abstract form *resolves* the forces that make finding one's way around in the space problematic. But the actual representations of the features, the patterns, and the contextual relationships between them, the pattern language, exist in a different conceptual domain to that in which the forces exist, a map is *not* the same thing as the 'space' it represents.

So the advantage that concentrating on context gives you is that it can be represented in the pattern language easily and accurately, and forces can't. Moreover it enables you to see that the real dynamics provided by the application of a pattern is not in building the solution, but in providing the context for the application of the next pattern. It does this by specifying the problem anew, restating the current constellation of forces. What you are working on is a *problem in a context* not a problem in the abstract. Using the context of the pattern in pattern language is a powerful way of sidestepping the need to understand the forces in the problem situation, or *even the exact context* of the problem. The pattern language is telling you that, at this point in the process, you really only have these options available to you. Thinking about anything else here is pointless. This means that you can analyse the situation in terms of the options that are actually pertinent. *Most of the context* of the problem, *most of the forces* at play can be ignored, because the important aspect of understanding the current stage of the process is provided by the pattern language. Applying one of the options alters the situation in a way that changes the problem.

If the context needs to be more closely specified then that should be done in the pattern language rather than in the patterns. This situation actually does arise in the novice programming situation. One of the main difficulties that novice programmers face is deciding what to do next. Part of the power of the pattern concept derives from the specification of context. A pattern says that faced with this situation, this is known to be a way of proceeding. But, in general terms, the proposition holds - a good part of the decision about which pattern to apply next is programming language dependent, so, as an example, it is not possible to apply a loop in any other Java context than a method. This implies that most of context is above the level of pattern detail, it would be a mistake to try to specify such choices in the patterns themselves. A much more natural way to illuminate context is to provide a map. This is what the pattern language does. Thus if there is a lot of contextual description appearing in a pattern then it probably indicates that extra patterns are needed on the map, that it is not detailed enough to make choosing a route obvious. A novice shouldn't have to read the pattern to make a choice *except in the case where there is no dramatic difference in terms of the final artefact between the choices.* The pattern form is about solving the problem, implementing the solution, not about discovering the context - you need a map for that.

6.6 The Dynamics of the Pattern-Problem Situation

The relationships in a system between the problem and the pattern language are a result of the need to carry out a task. You start with the system in a certain state ('Original State' in Figure 6.7), and the initial problem, and therefore, its context, is created by the need or the desire to do something. The forces that now pertain in the situation are the result of the impetus created by the need to change the state of the system in some way.

That is, the situation has been transformed into the initial problem and its context by the forces set up by the requirement for change. But, as Alexander points out, this is just too complex to analyse in terms of its forces (see (Alexander 1964, pp. 24-5)). The factor that makes design problematic is that the elements of a design are not independent - a simple list of the forces does not capture the dynamics of the relationships between them. Changing one aspect of a design

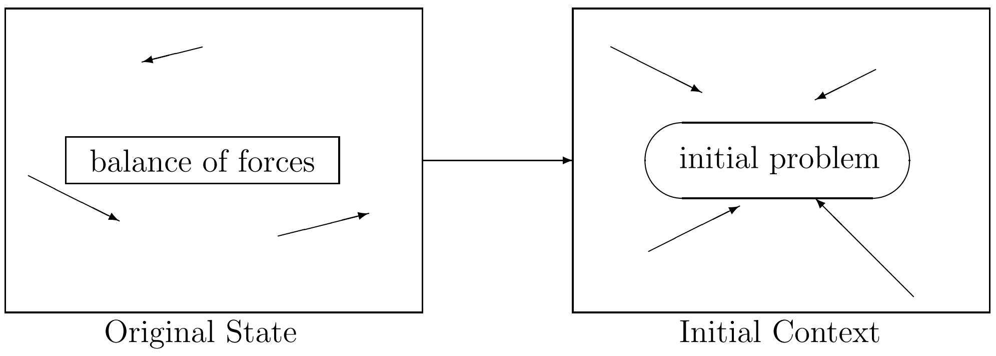


Figure 6.7. The initial problem sets up the context for the first pattern

will affect many others because the parts, and therefore, the forces, interact in complex ways.

> We know that we shall never find requirements which are totally independent. If we could, we could satisfy them one after the other, without ever running into conflicts. The very problem of design springs from the fact that this is not possible because of the field character of the form-context interaction.
>
> (See note 23 in (Alexander) 1964, p. 213)

What the pattern language brings to the complex is a view of the system that derives from prior experience in changing it. It says that these patterns have now come into play as a means of avoiding the need to closely analyse the forces pertaining in the context of the problem.

As a pattern is applied it changes the dynamics of the situation. The problem, its context, and the forces in it, are transformed by the pattern because, not only has the problem itself been modified, but the context of the problem has been altered by the interaction with the context of the pattern, that is, its location in the pattern language. Applying one pattern has brought the patterns below it into play. In other words the pattern has brought its context with it and thereby transformed the context of the problem. This is because the context of the pattern is a reflection of prior experience. In dealing with situations in the system people have discovered this particular structure of relationships between the patterns. So the pattern language structure is encoding the underlying order in a powerful way. It is, in a sense, a map of the common problems in the system. Any problem currently being dealt with will have its context in the system revealed by the pattern to which it is related. Prior experience relates problems to known solutions in the pattern form, and problems to their context in the web of relationships that forms the structure of the pattern language. "The language not only connects the patterns to each other, but helps them to come to life, by giving each one a realistic context, and encouraging imagination to give

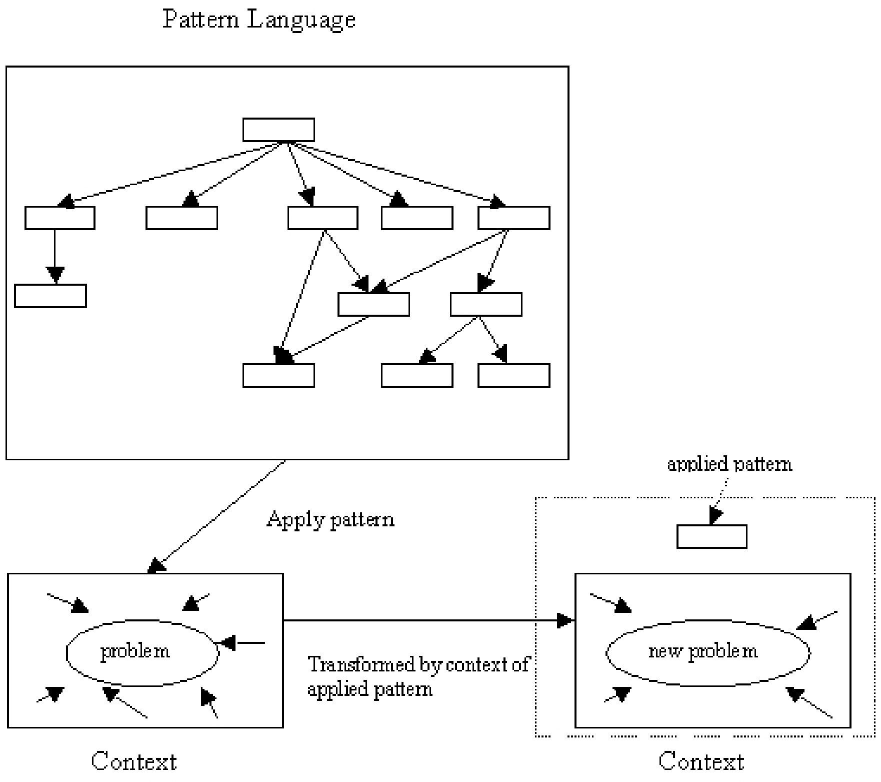


Figure 6.8. The applied pattern brings its context with it.

life to the combinations which the connected patterns generate" (Alexander 1979, p. 315). The pattern language process is an unfolding of the solution to the original problem through the dynamics of changing the problem and its context. In Alexander's terms, the *context of a problem* is the *configuration of the forces* in the situation. But it is simply too difficult to deal with the forces directly. Applying the patterns is thereby a way of avoiding this difficulty.

The advantage of the pattern language is that by illuminating the context of the pattern, its location in the pattern language, it locates the position of the problem in the overall structure of the system as well. Prior experience has located this pattern in this context, so the context and the forces involved in the situation currently being dealt with are likely to be similar. But you don't have to analyse the forces or understand the context of the problem situation itself, because the context of the pattern tells you which patterns have been found to be useful in dealing with this situation before, that have resolved these forces. The pattern language directs you to solutions without the need for close analysis of the problem context because in a vital way the problem context *is* the pattern context. The pattern is really just the problem with an exposition of its solution. The other patterns in the pattern language are similarly just the other problems common in the system. So it is only natural that the position of the pattern in the language tells you about the context of the problem. The pattern language is not just some artificial structure imposed on the system from the outside, it is an encapsulation of prior experience in the dealing with problems in the system.

By deciding to deal with the current situation by writing a program, you have implicitly added the programming language to the system in which the situation exists. The programming environment has necessarily become a part of the way that you now view the situation. And this is why the pattern language derived from the syntax of the programming language is relevant. It comes with the toolkit. Any situation has to be dealt with using programming tools, so the larger system is now constrained by what is possible and available in the chosen programming language. At its base level a programming language is just logic. So what you are really doing by bringing a programming tool to the system is to provide a means to logically analyse it. You are deciding that it is more efficient to deal with problems in this system using computers. Ipso facto you are now going to have to view all the problems in those terms. The pattern language of the programming syntax is now a part of the context of problems in this system.

This is advantageous because the toolkit has a history. Others have been using it to solve problems in other systems. But, more to the point, it was designed with solving programming problems in mind. This is why the particular paradigm of a programming language makes a difference. As we saw earlier, thinking of a program in object-oriented terms is quite different to thinking in procedural terms (Bergin 2000). So you are importing the prior experience of other people, including the language experts who designed it, and adding it to your own. This is all that a pattern language based on the syntax of the programming language is - a history of using these tools. There is nothing particularly modern in all this, of course. Thinking about any system in terms of a new technology changes the way that you think about it. Every advance in weapons technology changes the way that the conduct of battles is thought about because the new technology brings new patterns of use that can't be ignored in terms of overall strategy. The system now incorporates the patterns of use built into the new technology. And the experience of others becomes relevant because of this too.

6.7 Patterns and 'Wholeness'

The dependencies between the elements of a system are an expression of its 'wholeness', and the problematic nature of a 'design process' is caused by the fact that it is linear in nature - it is a 'program' in the general sense of that word. "The process of development is, in essence, a sequence of operations, each one of which differentiates the structure which has been laid down by the previous actions" (Alexander 1979, p. 371). So there is a fundamental conflict between the needs of the product ("wholeness") and the nature of the process that produces it.

> The designer as a form-maker is looking for integrity (in the sense of singleness); he wishes to form a unit, to synthesize, to bring elements

> together. A design program's origin, on the other hand, is analytical, and its effect is to fragment the problem. The opposition between these two aims, analysis and synthesis, has sometimes led people to maintain that in design intellect and art are incompatible, and that no analytical process can help a designer form unified well-organized designs.
>
> (Alexander 1964, p. 116)

If this conclusion, that "no analytical process can help a designer form unified well-organized designs" were to be true, then history tells us that there must be some non-analytical process that does work. It's difficult to accept that the examples of well designed 'artefacts' that we see all around us are all the product of pure chance. Moreover, the notion of design, considered widely, encompasses natural systems even though there is no 'conscious' process of design involved. In fact, it is 'natural' order that provides the clue to the mystery of the occurrence of good design in human affairs. Natural form is the product of processes that are, by definition, non-analytical - there is no 'conscious design' involved. This implies that nature is the model for good design, as Plato insisted. The order that we see in nature depends upon generality, the persistence of general features, Plato's forms, in spite of the everchanging flux that is process. These regularities are the factors in the universe that we call "the laws of nature" and represent Plato's statement of a theory of universal patterns (Watts 1982, p. 33).

What the pattern language gives us is a process that is fundamentally non-analytical in nature. In a sense the "analysis" has been done in advance, it is embodied by the order reflected in the language structure, which is itself, in turn, an expression of previous experience with the system. But remember, this structure is derived itself from process, the functioning of the system, and therefore reflects the "wholeness" of the system in a way that no analysis can accomplish. This is a very old idea, in fact it lies behind all attempts to explain the world whether they be religious or rational in spirit.

> According to Herakleitos, the one rational world-order is both unwilling and willing to be called by the name of Zeus ... i.e., we may speak of the orderly processes of nature in purely secular terms, or we may, if we wish, apply religious terms to them. But the important thing is to recognize their *objectivity*: the fact that they exist independently of our attempts to impose our own patterns upon them. The fire of the *kosmos* is both physical and metaphysical: it is not just change, but *ordered* change: the continuity of the general features of processes throughout unceasing change. The order is produced by the *Logos* or *metron*, which, in the last analysis, is physical or observable, and is "common to all things".
>
> (Watts 1982, p. 11)

In an important sense analysis is a sort of post facto approximation of reality. Analysis happens after the event. It aids "understanding" in that it provides the basis for predicting future events because of the fact that systems are ordered and therefore repetitive in nature, not through any inherent power of its own. Nothing is as thoroughly deterministic as the throw of dice, yet no amount of analysis or measurement of the details of the interactions will provide a means of predicting the result. The interactions occur, in a sense, without analysis or measurement of detail, so although the result is "determined" by the forces involved, mass, momentum, angles, velocities, and so on, these cannot be analysed in advance. Even if they could be "measured" this would still not be useful because the system functions, in effect, with the exact parameters involved, and any measurement, no matter how precise, remains an approximation. So even given measurements with an accuracy of a hundred million decimal points, this would not enable an accurate prediction because the system functions with an 'effective' accuracy to an infinite number of decimal points, that is, without any measurement as such occurring. Measurement lies on the semiotic side of the epistemic cut, functioning on the physical side (see Section 5.5).

Another way of saying this is that "understanding" is based on the order of the system, not the details themselves, because the system is expressed in the "continuity of the general features of processes throughout unceasing change" (Watts 1982, p. 11). All of our intellectual systems are built on general rules. The details are just data en mass, not information on which we can build understanding, or at least not the level of understanding that underlies the practice of a skill. Consider the difference between the 'art' and 'science' of riding a bicycle. The 'art' is exemplified by actually riding a bicycle while the 'science' is just an analysis of the various forces involved. The rider is doing the riding without ever having done the analysis, that is without understanding the 'science' in any intellectual way. The 'art' is based on practice of the skill, repetition, not just intellectual understanding, or even computation of the actual numbers involved.

> The principle by which the cyclist keeps his balance is not generally known. The rule observed by the cyclist is this. When he starts falling to the right he turns the handlebars to the right, so that the course of the bicycle is deflected along a curve towards the right. This results in a centrifugal force pushing the cyclist to the left and offsets the gravitational force dragging him down to the right. This manoeuvre presently throws the cyclist out of balance to the left, which he counteracts by turning the handlebars to the left; and so he continues to keep himself in balance by winding along a series of appropriate curvatures. A simple analysis shows that for a given angle of unbalance the curvature of each winding is inversely proportional to the square of the speed at which the cyclist is proceeding. But does this tell us exactly how to ride a bicycle? No. You obviously cannot adjust the curvature of your bicycle's path in proportion to the ratio of your unbalance over the square of your speed; and if you could you would fall off the machine, for there are a number of other factors to

> be taken into account in practice which are left out in the formulation of this rule. Rules of art can be useful, but they do not determine the practice of an art; they are maxims, which can serve as a guide to an art only if they can be integrated into the practical knowledge of the art. They cannot replace this knowledge.
>
> (Polanyi 1958, pp. 49-50)

Another example that Polanyi discusses is the 'touch' that pianists sweat blood to attain. Fame and fortune rest on this 'touch', which according to the science of the process of sounding a note on a piano, cannot exist (Polanyi 1958, pp. 50-1).

6.8 So what is a Pattern?

The description that best fits the pattern idea is that it is a feature of some system that forms a node in the concept hierarchy of a conceptual space. It is the fact that it is a node in the conceptual map, the pattern language, for that space that makes it useful in addressing the organisation of the system, not the simple fact of recurrence. Some recurrences are things to be avoided - mistakes, or anti-patterns perhaps - they do not contribute to the organisation or structure of the space. Or they even contribute negatively. It is all too easy to forget that a system is more than just structural form, that it is important because it contributes to a process, it *lives*, it exists as a 'whole' in its own right. So a pattern is a pattern because it contributes to a process that *structures* a conceptual space, it could hardly recur if that were not the case. So, in architectural design, "each pattern is an operator which differentiates space: that is, it creates distinctions where there were no distinctions before" (Alexander 1979, p. 373).

But on its own this is not enough. Buildings are only meaningful insofar as they are 'living space'. If patterns really are just about differentiating space then it's difficult to see how they generate the Quality Without a Name. It is entirely possible to envision space being differentiated in ways that do not feel good. And, indeed, isn't that the basis of Alexander's criticism of much modern architecture? "To me, this core fact about buildings in traditional societies being beautiful ... had to be accounted for, and the idea that buildings of our time, by comparison, were so oppressive and ugly, even the best of them, gradually emerged in my mind" (Alexander quoted in (Grabow) 1983, p. 37). There must be something else to be added to this definition. 'Each pattern is an operator which differentiates space in ways which create quality/feel good/contribute to the wholeness of life.' Patterns are about structuring the whole way of life of a community, about a process not an artefact. The built artefact is merely a factor in the larger process, a part of the superstructure of communal life. Just as a human skeleton is not an entity entirely in its own right, so a building is more than just a means of differentiating space. The skeleton provides the superstructure for human life. The built environment provides the superstructure for communal life.

Seen in this way the role of the architect/builder is more clearly analogous to that of the programmer. Both are dealing with systems - the architect/builder with the system of communal life, and the programmer, at a particular time, with a particular system that contributes to the functioning of a social or personal purpose of some kind, which is why the *program* is not the same as the *application* to which it is put, just as the physical building is not the "social function" it fulfils, or attempts to fulfil. In both cases the structural aspect is just a means to an end, the functioning of the system. The structure thus sits between two processes. The process of building it and the process that it helps sustain. Its 'wholeness' can only be defined in terms of the latter, not in structural terms alone. Pattern languages are a way of bridging the gap between the two processes. For a new structure to 'fit' into the already existing system, the way of life of the community, or the existing social or personal system, it must reflect, in some way, the pre-existing order. It is difficult to see how a new artefact can contribute to the 'wholeness' of a system unless this is true.

As an explanation of what patterns do, 'differentiating space' is too simple. In architectural terms, patterns transfer information about differentiating space in ways appropriate to the community system as a whole. They reflect social functional order more than they differentiate or structure space. This is important because it clarifies the role of patterns in programming. Patterns in programming are about the functioning of the system as a whole, not just its structure. A program is defined by what it does, not what it is. You don't care, in a sense, what the program is, as long as it contributes to the functioning of the system in which it is designed to be useful. The real symmetries lie in the two processes on either side of the program. This is, in fact, analogous to the situation with architectural/building patterns, but the analogy is obscured by the 'differentiating space' idea. The symmetries, the problems, the centres, the wholes, even the patterns, exist in terms of the system not just spatially. Of course, space *is* differentiated in the building process, so the spatial symmetries are a factor in architectural patterns. But the main thing being differentiated is the living organism, the community, the social relationships in fact.

Alexander's words about the role of patterns in 'wholeness of life' can only make sense if he is referring to more than just the structural aspects of building. Any building can only relate to life in terms of the organisation of living beings to which it relates in the larger communal system. Given this the analogy with software patterns is again clarified. The only life to which these patterns can contribute is seen, in these terms, as the larger system of which the software is a part. The software itself does not 'live'. Its 'life' is expressed in its interaction with the system seen as a whole community. Thus any analogies drawn between Alexander's discussion of aspects of pattern function must work at this level. Aesthetic considerations, for example, must be about the impact of the program in its expression, not just it's code form. The scale and impact of a building on human consciousness, the aesthetics in short, derives as much from its place in the human culture, as it does from its place in the physical environment. Even 'abandoned' buildings have a place in human culture, they relate to us through the continuous record of community that we call history.

So why are these things that we call patterns, patterns? What is it that is the same about all these varied constructs that we dare place them in the same category? In their physical reality or conceptual form they are all different. Some are doors, some are window sills, some are benches. Yet we refer to them all as 'patterns'. The 'pattern' aspect is the role they play in the larger organism of which they are superstructural elements. Repetition is one node of this 'pattern' relationship, and it gives the relationship its name. But the other aspect of a pattern is in its functioning, the way in which it resolves forces, relates structure to function. In physics, forces are uniquely a dynamic relationship in space. Some physical entity with size, shape and mass is changing in some way, or tending to change. Resisting change is as much a force as whatever it is that is being resisted. But it is important to see even simple existence as change. Time itself is a sort of force, part of the dynamics of existence. So each pattern is a resolution of forces.

But there are not that many different kinds of forces in the world. Therefore there will necessarily be a repetitive aspect in every manifestation of the resolution of the same forms in different circumstances. In the interaction of high frequency light with the normally functioning retina the resolution involves the universal that we call 'blueness'. No two circumstances that cause this 'blueness' are the same yet the essential nature of the reaction, the 'blueness', is always the same. The 'blueness' derives from the interaction, the playing out of the forces. The 'pattern-ness' of something is therefore a property of the interaction, not of any of the structural, physical entities involved in it. "The patterns repeat themselves because, under a given set of circumstances, there are always certain fields of relationships which are most nearly well adapted to the forces that exist" (Alexander 1979, p. 146) The patterns express themselves in structural entities, the physical and visible manifestation of the fields of relationships, but they are always essentially a dynamics of some kind, a resolution of forces.

This understanding of pattern-ness as 'resolution of forces' is fundamental to the idea of 'wholeness', of patterns being 'alive' or 'dead'. Any situation is more than just a collection of 'physical' forces. The physical environment is merely the superstructure of this mystery we call life. Life is a process. Think of a waterfall. It too is a process. The physical drop over the rocks, the water molecules themselves, these are all just superstructure. The waterfall is a visible manifestation of the process of resolving the forces of gravity and exclusion. The water molecules must move under the force of gravity but they cannot move to space that is occupied by rock, so the two forces resolve in the form of a waterfall. Any waterfall, no matter how different in appearance from other waterfalls we have seen, is the same in its resolution of forces - always. `WATERFALL` is the pattern that expresses the relationship between vertical-ness in physical topography and a fluid, water.

Every detail of a system can be seen as a force in that system in that it affects the functioning of the system. To exactly specify the flow of a waterfall at any moment I would have to know the position and velocity of every molecule in the entire system, in other words, I would have to deal with all the forces. But understanding a system at a fundamental level does not require it to be exactly specified. This is fortunate because the state of any *real* system can *never* be exactly specified, as, being based on measurement, specification resides on the subject side of the epistemic cut, and is therefore not properly a part of the ontological status of the system. Understanding is based, not on exact specification, but on the hierarchies of meaning that details form, patterns. If a level of detail is significant in terms of the functioning of a system then it will recur. There is a threshold in the configuration of forces that changes detail into pattern. All I have to know about the forces at work in the waterfall situation is derived from my previous experience - the pattern is obvious. Given any configuration of details, `WATERFALL` will always recur if the context of the situation is `RAINFALL` and `UNEVEN TOPOGRAPHY`. The pattern language of a system is simply the hierarchy of meaning within it, expressed during its functioning, it is a map of experience.

Life is the pattern of being alive. It is a resolution of the forces brought into play by self-organisation. A living form is distinguished from a non-living form by the presence of this force for self-organisation. The context of the pattern is physical reality, and the problem that the pattern solves, life, is the dynamic interaction between the drive for self-organisation and physical reality. One collection of molecules has formed into a larger entity that is self-organising, and in this way has differentiated itself from other collections of molecules that are either self-organising themselves, or not. But, like the waterfall, it is the process that defines the collection, makes it whole. It is an entity only because it self-organises. It's 'wholeness' is the resolution of the force for self-organisation against the forces of the physical nature of reality including the force that any molecule has to just exist, that is, to just be a molecule. When the force for self-organisation is gone all the molecules in the entity cease to be organised by the whole and revert to simple existence.

In a sense, life is the relationship between physical reality, time, and the tendency for self-organisation. A pattern in the built environment is a resolution of forces in this whole complex - but even more than this because there is another factor here. The environment is no longer just defined by the interplay of physical and biological forces but by mental ones as well. Some of these self-organising entities extend their organising ability into changing the environment in intentional ways. So there is now a larger self-organising entity, human community (society and culture). The patterns in a built environment are a resolution of forces in this whole complex, and will only be alive, or contribute to 'wholeness' to the extent that they do this successfully. Resolving forces, even successfully, that are just physical, forces that are just biological, or even forces that are from the combination of these two fields, is no longer going to be sufficient. Patterns in the built environment are about community, the life of the larger entity, and therefore resolve forces that are the result of relationships between all these fields.

Our human perception is approximating in the sense that it deals with representations of things (images, concepts, theories, etc.) rather than the things themselves. Thus it is in principle impossible for us to distinguish between our representation and the thing being represented - for our own cognitive purposes the representation *is* the thing. This is why it is not possible to compare qualitative experiences, not because the system that is giving rise to the experiences is different, but because the internal representations of the system are different. Presented with an automaton that fitted my representation of a sentient being I can do no more than compare it with my representation, and say that the two are, or are not, identical. This means that I can *never* say that an automaton that fits my representation *is* a sentient being. All I can ever say is that it fits my representation of such. This does not preclude making an assumption to that effect. As a working hypothesis we do this all the time. We assume from the fact that other people fit our representation of sentient beings that they actually are such, when in fact we have no real notion of their qualitative experience at all - we simply assume, based on our own experience, that it is similar to our own. Part of our everyday metaphysics is the assumption that our perception reflects reality. We are dualists from practical necessity - life just gets too complicated if you try to stick too rigorously to either realism or idealism.

We understand things by relating them. One way of doing this is categorisation - the relating of similar concepts. Another way is analogy. We 'understand' something by relating it to a different concept that we do understand. Both of these involve pattern languages. We are using the similarities between concepts (commonality) to increase order. Before we began to place plants into categories based on similar features, our view of the plant kingdom was a largely undifferentiated mass of detail, complicating even basic processes like identification of plants - the same plant species ending up with different names, or different species with the same name. Our understanding of the plant kingdom depends largely on our view of it reflecting the 'natural order' within it. The order of the whole system is based on the relationships between the structural order of different species. (Is the fact that one of the categories in the classification system is actually named 'order' a pure coincidence?). So, without a system of categories, the living world is just a vast complex undifferentiated mass of detail that is almost impossible to understand. Categorising things based on the similarities and the differences between them allows us to 'see' the patterns in the system. We have differentiated the mass; added coherence to the detail.

But an important aspect of patterns is their usefulness, not only in terms of understanding a complex system, but in maintaining and developing it, and it is at this point that the notion of a pattern as a tool for practical action becomes significant. So far we have discussed patterns primarily in terms of their effectiveness as metaphors, a means of bridging quite disparate systems, and, as shown in Table 6.1 there is an exact fit between pattern and metaphor at most levels. However practical action is not often considered in metaphorical terms, more usual are notions like templates and moulds, so the question arises about the fit between these concepts, mould and template, and pattern. Despite the superficial similarity, Table 6.1 demonstrates that there are fundamental differences at most levels of analysis.

	Pattern	Metaphor	Mould/Template
Is a form of	Repetition	Repetition	Repetition
Methodology	Generation	Generation	Replication
Works by	Prior experience	Prior experience	Strict definition
Produces	Understanding	Understanding	Exact copy
Transfers	Meaning	Meaning	Form
Process is	Informal	Informal	Formal

Table 6.1. Analysis of Patterns as Metaphors and Moulds or Templates

This analysis, in fact, highlights the power of the pattern idea, that it generates on the basis of the very essence of experience, that it is at the same time unique and yet familiar. Patterns are closer to metaphor in that they deal with meaning, not just form, as templates and moulds do.

6.9 Mind and Patterns

Patterns are the essence of mind and are the basis for its fundamental role in any form of creativity. But creativity can only happen through combination of existing entities, even if those entities are only ideas, so the *real* driving force of creation is pattern language. Faced with any situation in which new form is required, one first has to develop the 'language' of ideas that describes it. What makes a novel situation extraordinarily difficult is the fact that the 'language' to deal with it does not yet exist and has to be made up. Fortunately this existence of ours is not just that "continuous stream of sense impressions flashing past" that we described in Chapter 3 where "nothing ever repeats", there is, as the old saying goes, "nothing new under the sun", no situation is ever entirely novel. Some of the elements of the language that we are required to deal with any situation already exist, we have encountered them as patterns of our previous experience, and even if we haven't ourselves, other people may well have done so. Often, indeed, all the patterns we need in a novel situation exist already from our prior experience, the novelty really only exists as a sort of contextual illusion - the same sort of things are happening, or need to be made to happen, in a context in which we have not experienced them before.

So the language that we have to make up in a new situation consists, mostly, if not totally, of elements with which we are familiar in a different context. The language is *particular* to the current situation, but the components come mostly from the *general* pattern language of everyday experience. Think of how the particular pattern for `DOOR` could have arisen over time. As humans moved from living in natural shelters, such as caves, to manufactured ones, the problem of entry and egress was dealt with using the experience that caves have openings. `ENTRANCE` is thus a pattern that 'adds structure' to `SHELTER`, both of which derive from the the general patterns surrounding life in a cave. When the situation changes to life in an area devoid of natural shelters, these patterns are imported into the pattern language for the new situation. But, of course, once you start *manufacturing* shelters, a new language, derived from that for making tools, a 'language of manufacture' in effect, develops around the making of shelters. So, although the idea of a door is a novel one from the cave experience, once you have a language for making components of a shelter, walls and roof structures for example, the idea of making a component to close off the entrance when it is not being used, is not completely novel in terms of the 'making components of a shelter' language, it is just a different type of `COMPONENT`, a modification of `WALL` maybe. A language for any particular endeavour will thus consist mainly of patterns derived from languages for other types of experience, and the sum of all the pattern languages can be seen as a sort of general pattern language for the totality of life.

This means that a significant part of any creative process is establishing the particular language for it, in both directions, specialization and generalization. `ENTRANCE` in terms of the new language is a generalization of a component of the 'cave life' pattern language, but a `DOOR`, a pattern that 'adds structure' to `ENTRANCE` involves a specialization of the patterns of skill derived from making tools. But this is, in a sense, still a general language for building shelters. Any project for a particular shelter will be a specialization of this general language, and, as Christopher Alexander points out, it is this language-composing element that lies at the heart of design.

> So, the real work of any process of design lies in this task of making up the language, from which you can later generate the one particular design. You must make the language first, because it is the structure and the content of the language which determine the design. The individual buildings which you make, will live, or not, according to the depth and wholeness of the language which you use to make them with. But of course, once you have it, this language is general. If it has the power to make a single building live, it can be used a thousand times, to make a thousand buildings live.
>
> (Alexander 1979, p. 324)

The design of an novel artefact, is creation in the purest sense because once where there was nothing, there is now something. And before it could exist as an artefact it had to exist in conceptual terms. But when you are talking about something that does not yet exist, you are in the state of not really knowing what you are talking about. Dijkstra describes a situation of this kind that arose when he was part of a team designing a multiprogramming system.

> When the design is complete one must be able to talk meaningfully about it, but the final design may very well be something of a structure never talked about before. So the design team must invent its own language to talk about it, it must discover the illuminating concepts and invent good names for them. But it cannot wait to do so until the design is complete, for it needs the language in the act of designing! It is the old problem of the hen and the egg. I know of only one way of escaping from that infinite regress: invent the language that you seem to need, somewhat loosely wherever you aren't quite sure, and test its adequacy by trying to use it, for from their usage the new words will get their meaning.
>
> (Dijkstra 1982, p. 342)

But this means that you are working largely in the dark, for not only are the concepts unknown, they are constantly evolving. At the beginning of the design process one only has the vaguest idea of what the elements one is using to discuss the design will finally come to mean. This means the the design dialog involves 'words' that shift in meaning as the dialog progresses. No wonder the process of creation seems mystical. To an outsider the discussion of a team involved in design must appear to be verging on 'crazy talk'. Dijkstra had this feeling about the discussions of the team designing the multiprogramming system in which he took part.

> If during these discussions a stranger would have entered our room and would have listened to us for fifteen minutes, he would have made the remark "I don't believe that you know what you are talking about." Our answer would have been "Yes, you are right, and that is exactly why we are talking: we are trying to discover about precisely what we should be talking."
>
> (Dijkstra 1982, p. 342-3)

However, it is only possible for concepts to evolve out of shadowy origins like this if the language being used is informal. In a formal system the semantics of any given element is fixed, there is no such flexibility available, and this is why a programming language is the wrong language in which to *design* a program. Because you are constantly dealing with "concepts that are meaningless with respect to the original problem statement, but indispensable for the understanding of the solution" (Dijkstra 1982, p. 343) you have to make up the language as Alexander indicates, that is, you need a pattern language not a formal one. As Dijkstra says, this "is the only way I know of in which the mind can cope with such conceptual problems" (Dijkstra 1982, p. 343).

> Even Plato, who denounced the Milesian naturalists as atheists, nevertheless found in mind, the most remarkable of all natural phenomena, the distinctive characteristic of deity. And he did not shrink from the conclusion that deity, as mind, is not the "ultimate reality", because it is not causally independent. The "ultimate reality", in Plato's theory, is the world of the Forms - a doctrine which he emphasized, the "cognitively reliable", as Vlastos puts it: the domain of the logically necessary, by contrast with the world of contingent truth. In the doctrine of the Forms ... Plato was also trying to state the theory of universal patterns and standards, qualities and categories, and of kinds of real things (which he mistakenly represented as degrees of reality). Deity known to us as mind is causally dependent upon the Forms: a notion which we may interpret as the dependence of orderliness upon generality - i.e., that it is through the persistence of general features or characteristics that the rapidly changing processes of the universe retain their identity and exhibit the regularities in internal activities and in interaction which we describe as "general laws".
>
> (Watts 1982, p. 11)

What this boils down to is a view of mind as a way of arriving at a rational conception of the world. "A cognitive scientist would say that evolution constructed truth-finding cognitive processes" (Gopnik 1996, p. 489).

Categorisation is a classic 'patterning' activity that also involves a hierarchy of the connections between the categories, in other words, a pattern language. The phylogenetic tree is thus a pattern language for taxonomy, for tracing the evolutionary relationships among organisms. Again one sees the classic features of the pattern language, pattern (structural similarity, that is, repetition of form) and process (classification).

> There is and will remain a Platonic element in science which could not be taken away without ruining it. Among the infinite diversity of singular phenomena science can only look for invariants. There was a Platonic ambition in the systematic search for anatomical invariants to which the great nineteenth-century' naturalists, after Cuvier and Goethe, devoted themselves.
>
> Modern biologists sometimes do less than justice to the genius of the men who, behind the bewildering variety of morphologies and modes of life of living beings, succeeded in identifying, if not a unique 'form' at least a finite number of anatomical archetypes, each of them invariant within the group it characterized. It was of course not difficult to see that seals are mammals closely related to carnivores living on land. It was much harder to discern the same fundamental scheme in the anatomy of tunicates and vertebrates, so as to group them together in the phylum Chordata; and it was still more a feat to perceive the affinities between chordates and echinoderms; yet it is certain, and

> biochemistry confirms it, that sea urchins are much more closely related to us than the members of certain much more evolved groups of invertebrates such as the cephalopods, for example.
>
> (Monod 1974, p. 100)

But an even more critical aspect of categorisation is that no creativity is possible without it, precisely because what it sets up is a pattern language.

> Reasoning by similarity instead of calculation seems to have both evolutionary and parsimonious advantages and lies at the bottom of more complex reasoning. The ability to categorise is of fundamental value for the simplest train of thought. If a subject cannot identify and reidentify the object he reasons about, then he cannot entertain any continuous, coherent thoughts. He becomes a momentary individual. Categorisation develops in children at an early age from a general ability to notice different features of objects over context-bound reidentification to abstract categorisation. Abstract categorisation is in principle independent of detection of perceptual similarities and rests upon theoretical knowledge. It appears that conceptual or symbolic thought in this manner evolves from imagery.
>
> (Brinck 1997, pp. 12-3)

The important point about categorisation, indeed all the "truth-finding cognitive processes" that Gopnik and Meltzoff identify is that they contribute to the process of *understanding*. Carrying out any practical activity involves *understanding* the environment in which the activity is to be performed and the purpose to which it is directed. So the feature that makes a concept a pattern is its role in some process in terms of human purpose. This is the *critical* attribute of a pattern, that it *drives* process. In the learning to program situation, therefore, the process that the pattern language drives is programming, and the next chapter illustrates how the pattern language operates in enabling the process of designing the conceptual superstructure of a simple program of the sort typically encountered by novice programmers. But, before we do that, we need to consider the role of pattern language in the cognitive activity called programming.

6.10 Cognition in Programming

By its nature, programming involves high level cognitive activity. It is the process of solving a problem by creating a computer program, so it involves, not only understanding the programming language, but using that knowledge in creative ways. It is fortunate that the problem of levels of cognitive activity have been well formulated in general educational practice. The treatment of the notion of cognitive levels in learning can be used to help understand the cognitive load of other activities, such as programming. So, in this field, the work done on categorizing cognitive tasks embodied in Bloom's Taxonomy of Educational Objectives (the left-most column in Table 6.2) are applicable generally to the activity of programming itself, as well as specifically in the learning of programming as explored in Section 7.7.

A complicating factor in the interaction between the normal cognitive process and programming is the fact that there are two fundamental ways to approach the programming task. Each of these has implications for the overall development process as it impacts on the programmer and affects the artefact being built. Top-down programming is based on breaking down the original specification of what is needed into simpler and simpler pieces, until a level has been reached that corresponds to the primitives of the programming language to be used. The implications of this method for the cognitive process are that it mainly involves the first four of Bloom's categories. The fifth level, synthesis, which corresponds to overall program design, occurs implicitly, almost by default from the process of understanding and analyzing the task specification. The second style, bottom-up programming, requires the fifth level of Bloom's hierarchy, synthesis, right from the start, as it is based on building up more-or-less abstract components of the final system from the programming language primitives. In order to do this correctly, the assumption of a comprehensive grasp of what is required is implicit.

The implications for the artefact in the top-down approach are mainly the inbuilt rigidities involved in the overt concentration on the original specification. Modules built in this way tend to be rather task specific, and therefore less adaptable to changes in the original specification or to other situations. There is a temptation to provide globally exposed data structures and to share these between modules in a way that creates dependencies between functions that, ideally, should be isolated. Bottom-up programming naturally tends to produce modules that are more general, and thus more reusable, and having less dependencies. This implies more flexibility in responding to changes in the specification and to greater ease of testing and maintenance.

The trade-off between the two approaches, then, is that between the level of cognition required and program generality. Top-down is cognitively easier, but is likely to generate a program that has inbuilt rigidities. In practice, it mostly turns out that programming, in general, uses a mixture of both approaches, and this involves separating design from implementation. The design stage is basically top-down, but without involving any coding so that the design artefact is free of any implementation details. This means, in effect, that the lower cognitive level tasks involved in the top-down process, are directed at producing an artefact that encapsulates the programmer's understanding of the specification rather than the final program itself. It covers the assumption, which is implicit to the bottom-up process, that a comprehensive grasp of what is required exists in the programmer's mind, and so provides a sound basis for that approach

What this real world situation reveals is the intuitive fact that, given a complex task, people will tend to try to reduce the cognitive load by tackling the lower level cognitive activities involved in understanding it, first. The danger of separating design from implementation, however, is that the concepts used in the design may not easily map onto the details of the programming language used in implementation. Clearly, the idea of reducing the cognitive load is particularly pertinent to novice programmers, so it is important in the learning situation to facilitate this. However it is also important that the concepts used in doing this are not completely abstract in terms of the programming environment. A pattern language for a system is directly derived from the workings of the system, it is an expression of the system's underlying order. Using a pattern language for the design stage therefore allows the top-down process to occur using concepts, the patterns, that are still firmly rooted in the environment that is involved in the bottom-up process. The synthesizing aspect is built into the decomposition tool by the fact that the patterns are elements in a generative system, the pattern language, but are nevertheless more general and abstract than the raw programming language concepts.

Cognitive categories	Programming	Pattern paradigm
Knowledge	Programming constructs	Pattern name
Comprehension	Programming construct semantics	Problem
Application	Use of programming syntax	Solution
Analysis	Understand the problem	Pattern Language
Synthesis	Design the solution	Pattern process
Evaluation	Coding, testing and debugging	Code Examples

Table 6.2: Bloom's Categories in Programming Terms. Adapted from (Bloom et al 1971)

Table 6.2 shows how we think the activity of programming breaks down into the cognitive categories identified by Bloom, in general terms and in pattern terms. The first three categories involve the features of the programming system, and these are covered in the pattern form itself, meaning that they can be effectively 'ignored' by using the pattern language in the Analysis and Synthesis levels. This is the 'automatism' of the expert discussed in Chapter 9. In effect, the language features are being represented in the process of designing the conceptual solution to the original specification of the problem simply by the name of the pattern. The conceptual design involves analysing the problem and synthesising the solution, and this revolves around the use of the pattern language in the pattern process described in Chapter 7. But, most significantly, the pattern way of looking at programming reveals that coding the solution can be seen as a way of evaluating it. In a sense, the code form 'tests' the validity of the conceptual solution, the purpose of the program as expressed in the original specification is shown to be either expressed or not expressed in terms of a computing system.

It turns out, then, that the pattern language idea combines the advantages of the top down and bottom up methods by hiding the low level details required in the bottom up process in the pattern form. That this is a valid way of viewing programming is shown by the fact that this is how the ‘automatism’ of the expert proceeds. The expert bypasses the need to think about the low level detail by having thoroughly internalised the pattern language. So, the apparent ‘paradox’ of the expert programmer, the ability to design and code at the same time is ‘explained’ by the ‘revelation’ that coding, although involving the use of low level programming constructs is actually part of the high level cognitive activity of Evaluation! But, and this is the really significant point in terms of programming pedagogy, it illuminates the exact source of the difficulties that novice programmers exhibit. By attempting to code the solution they are tackling the highest level of cognitive process *without* any appreciation of the hierarchical ordering of the conceptual domain, not the ‘automatic’ implicit language built up over years of programming practice, nor any explicit representation given to them by their instructors. No wonder it’s like trying to fly without having grown wings first. Just as nobody is born with wings, nobody is born with a pattern language for programming either.

Chapter 7

The Pattern Process in Action

In what we call thinking the mind isn't "directed" but suspended. You don't give it rules. You teach it to receive. You don't clear the ground to build unobstructed: you make a little clearing where the penumbra of an almost-given will be able to enter and modify its contour.

(Jean-François Lyotard)

7.1 Learning to Program

The factor that makes learning to program so difficult is that there are multiple activities involved. Firstly, the novice is attempting to build a mental representation, a model, of the programming system, that is, to acquire the basic knowledge of the programming system that is required to use it. Secondly, she is trying to use that model to develop a program. However because the program addresses a situation in a domain other than programming, designing it involves understanding the situation as expressed in the task specification, so a third strand in the complex of activities is translating between the conceptual domain in which the task exists and the programming system. Moreover, as we will see in Chapter 8 all these various processes are themselves made up of a complex weaving of memory and meaning expressed in the transformation from static declarative knowledge to its procedural form. The most surprising aspect of all this is that *anyone* ever learns to program. Clearly this situation is a recipe for massive cognitive overload.

This is particularly so because the method that educators are relying on to drive the transformation from static to dynamic knowledge is the practical component, the experience of writing programs. Despite the attempts to bootstrap this transformation, that is, to start with simple examples, it is clear that it is this factor that lies at the heart of the difficulties expressed by novice programmers. Most novices are able to demonstrate a reasonable grasp of the basic programming language features, but many fail to be able to develop even the simplest program (Lister & Leaney 2003, p. 221). So the problem faced by educators is that of providing a means of facilitating the transformation from static to dynamic knowledge. Like Helen Keller's teacher we are confronted with the problem of an incomplete conceptualisation in the minds of our pupils.

However, there is a more fundamental problem here, and that is that the static, declarative knowledge, the basic set of features of the programming system, is presented in a form that is pertinent to the operation of the execution process in the machine, not the transformation from static to dynamic knowledge in the programmer's mind. So there is a category or ontological error involved here. The transformation process is based on understanding, but the execution involves pure mechanism. Whatever the properties of mind are that are involved in understanding, they are being incorrectly ascribed to the programming system. These two processes, understanding and execution, involve different processing systems, language and logic, the sequence of steps taken in each is not the same, and therefore they are, or *should be*, based on different combinatorial systems.

What this boils down to is the statement that a programming language is not the right language for the development of the conceptual understanding of a task in the domain in which it exists that leads to its incarnation as a process to be executed on a computer. This has long been recognised at the advanced program development level, hence the use of 'modeling languages' in the conceptual design stage in Software Engineering. Unfortunately, the particular 'modeling languages' developed for use at this level are not suitable for programs at the novice level, nevertheless the same situation pertains. In any case, the process must end up producing an artefact that does, in fact, run on a computer, so there must be some sort of bridging of the gap at some level. In the advanced situation the two sides of the gap, the development of the conceptual solution and the coded version, are usually handled by different groups of people, experts in using the modeling language to develop the solution in model form and experts in translating from the model to the code form.

Clearly, in the case of learning to program, the novice programmer has to perform both roles. Since the roles can't be separated at the physical level, they must be separated at the temporal level, that is, one should not attempt to perform them simultaneously. What is needed is a mechanism that separates the two tasks without rendering them meaningless to each other. That is, a conceptual solution that is not expressible in the execution of a machine is just as unacceptable as an executable solution that does not perform the task required by the initial specification. The bridging mechanism is therefore a way of thinking about the task in terms of the execution of the machine that *does not* constrain the thinking to just the execution level operations. A solution is *not fundamentally creative* in terms of machine execution as the basic operations that a computer can perform are set, therefore it actually exists only at the conceptual level because execution is pure routine, it requires nothing that is really new at execution level.

If "thinking is the ability to contemplate something in its absence" in

> Kosslyn's words, then creativity is the ability to contemplate something that has never existed before. This applies to the artist who creates new visions, the writer, the composer, the scientist who conceives of a new theory, the mathematician, the architect, the inventor - in short, anyone who goes beyond the routine in his or her profession. We value creativity as one of the most significant achievements of the human brain. Can we say anything about possible mechanisms?
> Again, we are tempted to look for this function at the highest cortical levels. But it is instructive to examine the procedure employed in some of these creative acts. An artist may have an idea for a painting, but he may not immediately go to the canvas. Instead, he often begins with a series of sketches.
> But why sketch? Why is it necessary to externalize the idea conceived in the brain, and then have the brain examine it? ... In every creative act we observe this bootstrap process in which nascent ideas are externalized and then taken in again by the brain to be reexamined and modified in a creative loop.
> But sketchpads ... are relatively recent acquisitions to aid our creative activities. If the picture I have drawn of the seeing and perceiving brain is correct, then similar processes are built into our brains.
> (Harth 1993, pp. 74-5)

The point about programming, then, is that although program execution is performed in programming system statements (made in brush stokes), the thinking occurs at the level of ideas, not such statements (brush stokes). And just as the externalising of thinking aids the creative process in composing a painting, so too does externalising the concepts involved in designing a program. Just as trying to envision the finished painting entirely in the brain is too much for all but the most highly experienced painter, so trying to develop a program by *mentally executing* it equally restrictive.

> The internal sketchpad has its limitations. ... The images drawn on it are ghostly and evanescent. It is difficult to hold complex patterns in our mind for long and subject them to detailed scrutiny. This must have been a severe limitation to our earliest creative drives. We can now appreciate the enormous advantage humans gained when they invented (or discovered) the ability to complete the projection, that is, to externalize their mental images beyond the sketchpad of the [mind] by creating permanent images in the world around them.
> (Harth 1993, pp. 74-5)

Especially for novices, this externalisation of the language of ideas, is critical in terms of assisting the creative process. Learning to program involves the application of programming system features to the solving of novel problems, so the features will be expressed in the bridging mechanism, but in conceptual rather than executable form. We are dealing with the pattern of the solution not the process of execution, and this is why we need a pattern language. Design is a different process and the pattern language concept is fundamentally about providing a mechanism for driving the design process. Patterns are not just expositions of a solution to a problem, they provide a generative framework, and this is the purport of the 'language' idea. The power of the pattern concept is based on the solving of problems using known solutions. Problems tend to recur in many different situations, and it is the repetition that makes them patterns that allows them to be used as a basis for design. The software pattern concept is based on this simple fact of recurrence. Since the problem-solution combination keeps recurring, the idea is to design an artefact, the pattern, that captures its essence in a standard form. However, since most programming problems are too large to be solved using a single pattern, the pattern concept must include the facility for patterns to be put together to form a 'pattern sequence', that is, a particular configuration of patterns that solves a particular problem.

Fortunately, the patterns in a particular domain form a powerful network of connections in terms of the context in which they appear. For example, if you are building or designing a wall you need to think about other things, like doors, and maybe even windows. This network of relationships between the patterns in a domain gives the overall collection a degree of coherence and forms the structure that is known as a pattern language. The use of the word 'language' is suggestive here because of the contextual meaning implied by the relationships between the patterns. Solutions for larger problems can be constructed from the individual patterns because of the connections of meaning between them, just as the relationships between words allow the construction of larger concepts through combination. What transforms the collection of patterns in a domain into a language, is the relationships between the patterns. It is from this network of relationships that the process derives, and it is the process that reveals the fundamental properties that give order to designs, "order as becoming" as Alexander says (Alexander 2002*a*).

A pattern language, then, represents the design space as a network of the patterns, and therefore the problems, that occur within it. It is the web of meaning and provides the source of the process of design. The language provides the dynamics for generating designs just as a natural language provides the dynamics for generating written artefacts. It provides a means for developing sequences of patterns.

> To get it [a sequence], a static pattern language ... was then re-stated as a generative sequence. In its sequence form it shows the user the process of unfolding, in sequence, in such a way as to allow a good building to be made, very easily, for the particular conditions of a given site.
>
> (Alexander 2002*b*, p. 303)

In approaching a programming problem, the idea is to analyse the context of the problem with a view to understanding the forces or constraints that will shape the solution. Once this is done, the forces so exposed provide the means to begin construction of the 'pattern sequence' for the solution. The advantages of the pattern approach are that the configuration of forces in a given problem situation can suggest which pattern best matches, thus providing an element of synthesis within the analytic process, and that the 'pattern sequence' for the solution can be constructed without having to deal with the code directly. The pattern language drives the sequence-producing process, the product of which, in turn, specifies the coded solution. In a sense, the pattern language is used to translate the natural language specification of the problem into a conceptual form, the sequence of patterns that solves it, and it is this concept level description that drives the production of the code, not the original problem specification.

So the 'language' notion is important in terms of driving the design process but it seems that this is often overlooked in practice. In the migration of the pattern language concept into the computer science field, the emphasis has been on the reuse of proven solutions, which in programming terms, are previous experience packaged in pattern form. But as Alexander himself points out, the adoption of his thinking by the Computer Science community has been incomplete. At the OOPSLA conference in 1996 he was invited to address the several thousand attendees, and delivered what, reading between the lines of convention and politeness, can be seen as a fairly devastating criticism. He said that he could see lots of patterns, and that was nice, but he could see no sense of moral purpose in what was happening in Computer Science, and worse, he could see no generative structural order - no process or language.

> What, now, of my evaluation of what you are doing with patterns in computer science? (Bear in mind, as you hear my comments, that they need to be taken with a grain of salt; I'm ignorant; I'm not in your field.) When I look at the object-oriented work on patterns that I've seen, I see the format of a pattern (context, problem, solution, and so forth). It is a nice and useful format. It allows you to write down good ideas about software design in a way that can be discussed, shared, modified, and so forth. So, it is a really useful vehicle of communication.
>
> (Alexander 1999)

He then goes on to point out that his own work in pattern languages was about generating wholeness, morphological and moral coherence, not just providing a "useful vehicle of communication," and asks if these aspects "have yet been translated" into computer science as this is the core issue of the pattern language idea (Alexander 1999).

Alexander's argument is that, in essence, the pattern idea is a way of looking at a system. As a collection of related concepts organised in a way that is useful in terms of some purpose, a system has both structure and purpose, and the pattern view reflects this fundamental wholeness, the "moral coherence" that Alexander discusses and which we explore in Section 11.4. The patterns are a representation of the structural detail, and the pattern language diagram is about the overall order of the system and the process of building and maintaining it. This is the advantage of patterns over other methods of designing a program, such as pseudocode, flowcharts, Nassi-Schneiderman charts and the like. In these cases the solution is generated by the programmer rather than being, partly at least, driven by the relationships between the concepts that are illuminated by the pattern language. The pattern language is a more powerful abstraction. It doesn't just provide a set of symbols to be manipulated by the designer; it adds a view of the context, the relationships between the concepts, to the design space.

7.2 The Pattern Process

The next two sections, 7.3 and 7.4, of this chapter are based on papers presented at the Third Asia-Pacific Conference on Pattern Languages of Programs, KoalaPLoP 2002 (Porter & Calder 2003*a*), and the Fifth Australasian Computing Education Conference in 2003 (Porter & Calder 2003*b*). These sections, 7.3 and 7.4, attempt to demonstrate how the process of designing a solution to a programming problem derives from the use of a pattern language, but none of the elements used, the problem, the patterns, or the language, are intended to be realistic in terms of pedagogical practice - the idea is to simply demonstrate the pattern process. To do this we introduce a simple pattern language based on the C programming language as it might be presented to novice programmers. Pitching the explanation to this level is justified on two grounds. Firstly, designing a pattern language around just higher level programming constructs such as those introduced in the GOF book is difficult, maybe impossible. Indeed the authors state that "it's hard to see how we could provide a "complete" set of patterns, one that offers step-by-step instructions for designing an application" (Gamma, Helm, Johnson & Vlissides 1995, p. 357), which is precisely the role that Alexander assigns to his notion of 'language'.

> As in the case of natural languages, the pattern language is generative. It not only tells us the rules of arrangement, but shows us how to construct arrangements - as many as we want - which satisfy the rules.
>
> (Alexander 1979, p. 186)

Because the higher level constructs explored in the initial enthusiasm for software patterns do not 'naturally' form a language in terms of the connections between them, this backbone of connectivity has to be provided by some other means. This backbone can only be constructed, via the ultimate *purpose* of programming system features, that is writing programs, and because no program can be written in anything but the primitive features, it is clear that the *patterns of experience* must connect at that level. After all, even the advanced level features can only be expressed in terms of the machine at the level of system primitives.

The point is that the higher level patterns *do*, in actual fact, connect to the backbone, but appear disconnected if the backbone is absent. In a sense, this is a justification, apart from any pedagogical purpose, for patterns based on the basic features of a programming language - no pattern language incorporating patterns at a higher conceptual level, such as the GOF patterns, is possible without them.

Secondly patterns are primarily instructional devices - a "pattern is the distilled result of the experiences of experts" written "for the benefit of both novices and experts" (Nelson 1999, p. 364). But the main difficulty that novices experience is precisely that involved in putting concepts together. Expert programmers already know the process, they don't explicitly require the generative power of a pattern language because they have, in a sense, internalised it. The high level patterns don't need to be presented in pattern language form because if you can understand them you are already a competent programmer and so you just fit them into your existing understanding. But this is not possible for novices, so the pattern language form is critical at this level.

Because it requires the learning of a skill, most educators involved in teaching programming agree that many students struggle in this field.

> Results from a recent project by McCracken et al. (2001) are compelling, because of the number of authors from differing educational institutions and cultures. The 10 authors teach introductory programming across 8 universities, in 5 countries. Each author tested his/her own students on a common set of programming tasks. The students performed much more poorly than the authors had expected. The students did not simply fail to complete the set task, most students did not even get close to solving the task.
>
> (Lister & Leaney 2003, p. 221)

But concepts and meaning are the very foundations of human intelligence, so we are looking at a misfit between the way we think and the way that programming is taught - it's just a particular form of problem solving, after all. There have been many studies into the relationship between programming and problem solving ability (Mayer, Dyck & Vilberg 1986) (VanLengen & Maddux 1990) (Reed 1998) (East & Wallingford 1997) and the similar relationship between mathematics and inductive reasoning (Haverty et al 2000), and these concerns have tended to flow into the idea of using patterns in introductory programming because of the resonance of the apparent cognitive difficulties exhibited by novices in building solutions with the problem-solution pair aspect of patterns.

> Much psychological research ... suggests that programming expertise is partly represented by a knowledge base of pattern-like chunks, variously named plans, templates, schemas, or idioms. ... Components of such a chunk resemble those described in Design Patterns. ... Additional research suggests that students gain expertise in programming and other disciplines from a process of knowledge integration.

(Clancy & Linn 1999)

The suggestion, here, is that patterns are useful in the integration of knowledge, in the task of putting disparate concepts and ideas into a coherent form, and this should be advantageous to novices. So the motivation for this chapter is an examination the process that arises from the use of a pattern language on the basis that specifying the benefits that may be expected in the novice programming environment will demonstrate the power of the pattern language idea generally in software development. In its original setting, building architecture, the pattern concept always had a strong sense of process built into it, but this seems to have been lacking in the software pattern field (Alexander 1999).

7.3 Applying Patterns to a Problem

Novices need to be given a clearly defined notion of process because most of the difficulties at this level arise from an inability to apply knowledge, rather than from a lack of knowledge. It is this factor that indicates the significance of the 'language' aspect of patterns in the educational context. The pattern form associates a problem and a solution and analyses the forces that it resolves, encapsulates the knowledge, and the language diagram provides information about the context in which the pattern is applicable, the operational semantics. This suggests that analysing the problem situation in these terms should provide a way of thinking about the problem in terms of patterns, which should help point towards a solution. Section 7.4 demonstrates the use of this process in the solving of a simple programming problem. Figure 7.1 illustrates the development process. The steps contained in the dashed rectangle attempt to identify the pattern in the catalogue of patterns that 'best fits' the context and forces of the problem. Applying this pattern modifies or refines the initial problem as shown in Figure 7.3.

One of the difficulties for experts in instructing novices is that the expert will make unconscious assumptions about a process like this. The expert recognizes the patterns in a situation almost without thinking, and therefore doesn't explain the entire process when teaching it. As Sleeman pointed out (Sleeman 1986), most people have great difficulty in explaining the process of finding the largest of a set of integers because they do it without consciously thinking about how it actually happens. In this case the expert, and even novices as they progress, will tend not to have to explicitly iterate through an entire catalogue of patterns in order to match a pattern to a problem. Instead the pattern selection processes will tend to merge into a single step, "choose the pattern that best advances the solution of this problem". Merging the dashed box step, "choose pattern", with the next step, "apply pattern", creates an overall process to be called 'add pattern' in Figure 7.3.

There are three elements to the problem solving process. The first is the

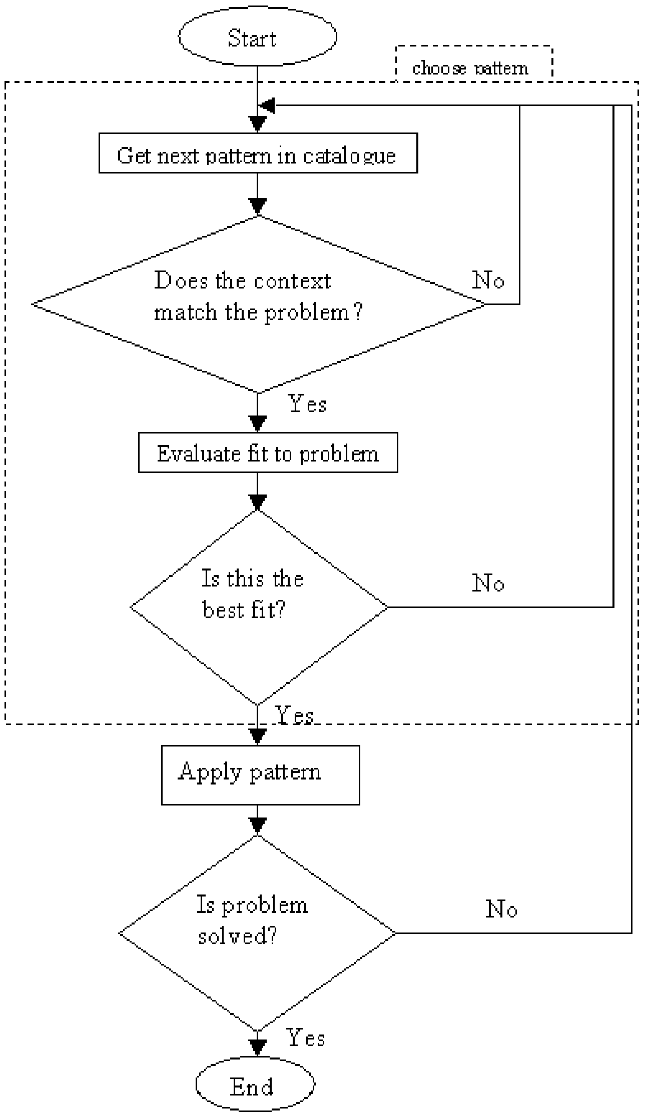


Figure 7.1. Building the 'pattern sequence'

pattern language diagram (see Figure 7.2), which provides the context for the next pattern to be applied. An indication of the forces resolved by applying it is garnered from the 'remaining problem' section of the 'add pattern' process illustrated in Figure 7.3, which is the second element. Further details about applying the pattern are available, if necessary, in the pattern itself. We have not included the patterns here as the pattern detail is largely irrelevant to our purpose of demonstrating the pattern process that is driven by the contextual representation - the pattern language. The third element, the context for the next pattern, is derived by following the arrows in the pattern language diagram downwards from the previously applied pattern. A single solid arrow means that the pattern to which it points will always be applicable, while arrows with dashed lines indicate that the context involves making a choice between several patterns. The semantics of any downward pointing arrow is that following it to its target 'adds structure' to the pattern at its source. A pattern will remain open until

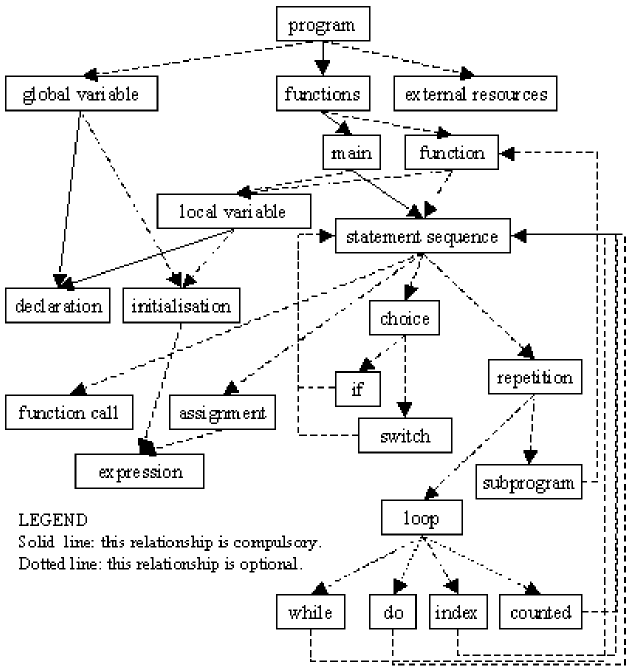


Figure 7.2. The Pattern Language

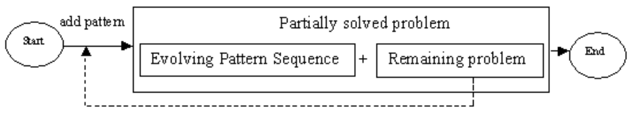


Figure 7.3. Adding a pattern to the sequence

there is no more ‘structure’ to be added to it, at which point it becomes complete. Note that some arrows lead back to patterns higher in the diagram, indicating recursion in the language. For example, a loop will have a block of code that is to be repeated. This block needs to be built up from `STATEMENT SEQUENCE` like any other, and the upwards arrows cover this situation.

The process of solving a problem, therefore, takes the following form:

1. Ascertain the forces in the unresolved part of the problem from the ‘remaining problem’ section of the ‘add pattern’ process.

2. Find the context of the next pattern in the pattern language diagram.

3. Apply the pattern with the help of the details in the pattern itself.

This interplay between the three elements drives the process that results in the building of a sequence of patterns that describes the solution.

Identifying the context of a problem is a matter of knowing the location of the last applied open pattern (that is, the pattern that still needs structure added to it). This is an indication of where you are in the development of your solution. The use of context, therefore, involves following the arrows from previously applied patterns. That is, the context for a pattern to be applied is a pattern that is still incomplete, that still requires additional structure added. In this case it is `STATEMENT SEQUENCE` again, unless it is clear that the current sequence is complete. If it isn't, the context is the same as before, the four patterns below `STATEMENT SEQUENCE`, and we then refer back to the current state of the problem solution to study the remaining problem for guidance in making a choice between them. This shows that in applying the 'add pattern' process in Figure 3, the remaining problem is the source of an indication of the forces in the situation, and the position in the pattern language provides the context for the next pattern. The movement back and forth between the two diagrams is the relating of forces and context, and this is what drives the problem solving process, building the 'pattern sequence'.

The subsequent steps repeat the process of selecting and applying further patterns until the solution is completely specified and no more refinement is necessary. The pattern sequence for the solution has thereby been constructed. Note that the process does not specify when coding is done. It can be done at any stage if the solution up to that point needs testing. Otherwise it can be left until the pattern sequence is complete.

7.4 A Simple Example

This section works through the process of solving a simple problem using patterns. It attempts to enumerate those things that are always done when solving a problem. That is, we are looking for the activities that are common to the problem solving process. So in that sense we are building a pattern view of the problem solving process.

The Problem Specification.
Write a program to produce a multiplication table for integers from 1 to 9 as shown below.

```
1
2  4
3  6  9
4  8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
8 16 24 32 40 48 56 64
```

```
9 18 27 36 45 54 63 72 81
```

Producing the above output on the computer screen is the initial problem. Applying the process illustrated by Figures 1 and 3 to this initial problem will divide the problem into two parts, the solved and unsolved part. The unsolved part, 'remaining problem' in Figure 3, is then put into the 'add pattern' step to again divide the problem. Continued iterations of this process will generate a sequence of patterns that 'describe' the solution. The iterations continue until such time as there is no more 'remaining problem'. The remainder of this section is a description of this process. Some iterations are insignificant in terms of understanding how the process works, and are subsumed as elements of one larger iteration. In these cases the new 'pattern sequence' diagram produced will contain several new patterns instead of the normal one.

7.4.1 First iteration of 'add pattern' process

The problem specification states that the task is to write a program to produce some output on the screen. Thus the first pattern to be indicated is `PROGRAM`. This pattern indicates that it is the appropriate choice when the problem is suitable to a computer based solution and it specifies the structure of a program in terms of C syntax.

Pattern 1 : `PROGRAM`

7.4.2 Second iteration of 'add pattern' process

Referring to the location of `PROGRAM` in the pattern language diagram gives us the context for the next pattern, which involves a choice between three possibilities, one of which, `FUNCTIONS`, is compulsory. In a situation like this, where there is a combination of optional and compulsory lines of development, the optional ones should be investigated first, and referring back to the problem specification suggests that some external resources in the form of the output routines in the input/output library are all that is indicated by these. Taking the compulsory arrow to `FUNCTIONS` involves making a decision about the nature of the program. Thus, so far, the solution involves a sequence of three patterns.

Pattern 1 : `PROGRAM`
Pattern 2 : `EXTERNAL RESOURCES`
Pattern 3 : `FUNCTIONS`

7.4.3 Third iteration of 'add pattern' process

The patterns in the context formed by `FUNCTIONS` are `FUNCTION` and `MAIN`. In reading the 'remaining problem', no need for a separate function suggests itself at this stage. This leaves `MAIN` where the actual execution of the program will begin, thus involving the next pattern `STATEMENT SEQUENCE`. The patterns applied so far are the mainly mechanical actions that set up the programming environment, and applying them has not materially advanced the development of the solution. `STATEMENT SEQUENCE`, however, sets the context for the attacking of the problem, and offers four ways of doing this. Making the choice between the four patterns below `STATEMENT SEQUENCE` means making an analysis of the forces in the remaining problem for clues.

Pattern 1 : `PROGRAM`
Pattern 2 : `EXTERNAL RESOURCES`
Pattern 3 : `FUNCTIONS`
Pattern 4 : `MAIN`
Pattern 5 : `STATEMENT SEQUENCE`

7.4.4 Fourth iteration of 'add pattern' process

The repetitive nature of the output required is the force acting in this situation, which suggests that the program control should consist of repeated action. Fundamentally the task is to generate a specific set of output lines, corresponding to the rows of the table, This problem is addressed by the `REPETITION` pattern, which tells how to perform a series of actions, in this case "print a line", several times in succession. `REPETITION` offers a choice between `LOOP` or `SUBPROGRAM` and the pattern indicates that, as the repetitions are, in this case, contiguous, `LOOP` is the choice we should make. The `LOOP` pattern shows how the division of the program into a repeating and a non-repeating part is represented in a sequence, and gives several variants. In this case, the best match is with the index-loop form, which is appropriate when the action to be performed varies for each repetition, and the variation can be expressed by a series of values. Specifically, the loop will execute for each of the values, 1 to 9, each line displaying multiples of those values.

For this problem, the entire task is contained in the code to"print a line"; there is nothing left of the original problem that is not repeated. Reference to the `LOOP` pattern tells us that therefore the rest of the sequence will be contained inside the loop as there is no non-repeating part to go after the loop. The box in the `LOOP` pattern contains the repeating block - if the remaining problem suggested any need for further non-repeating code, the pattern sequence for this would continue after the box.

Pattern 1 : PROGRAM.
Pattern 2 : EXTERNAL RESOURCES.
Pattern 3 : FUNCTIONS.
Pattern 4 : MAIN.
Pattern 5 : STATEMENT SEQUENCE.
Pattern 6 : REPETITION.
Pattern 7 : LOOP.
Pattern 8 :

INDEX LOOP
remaining problem (print columns)

7.4.5 Fifth iteration of 'add pattern' process

The repeating part of the problem will involve setting up a separate sequence of C statements contained in the block of code that is to be executed multiple times. This fact is indicated by the arrow that leads upwards to STATEMENT SEQUENCE, from where the repeating block can be developed. This provides the context for the first pattern in the repeating block. The remaining problem shows the need to examine what is required for each print line statement in the loop. The line is made up, conceptually, of a series of columns. Just as the LOOP pattern provided the answer to repeating rows it is clear that repeating columns probably requires a similar approach. That is, each line will be a series of columns printed out by an index loop. Proceeding through REPETITION and LOOP to INDEX LOOP is the same sequence as before, so here it is telescoped into the one pattern, INDEX LOOP.

This gives us the overall structure of the solution, which now takes the form shown below.

Pattern 1 : PROGRAM.
Pattern 2 : EXTERNAL RESOURCES.
Pattern 3 : FUNCTIONS.
Pattern 4 : MAIN.
Pattern 5 : STATEMENT SEQUENCE.
Pattern 6 : REPETITION.
Pattern 7 : LOOP.
Pattern 8 :

INDEX LOOP
Pattern 9 :

INDEX LOOP
remaining problem (print cell)

7.4.6 Sixth iteration of 'add pattern' process

Hence we have reduced the problem to "print cell", which in turn reduces to the problem of identifying and producing the number that makes up each cell. In a multiplication table, each cell contains the product of the row number and the column number. That is, each cell contains the formula `row * column`. In this case the loop control variables directly represent these values, so the "print column" sub-problem becomes:

```
print (outer index * inner index)
```

One obvious feature of the required output that has not been considered yet, is the fact that each successive line increases in length by one column. The number of columns in each line is constrained by the line number.

```
for (j = 1; j <= i; j++)
```
where j controls the inner loop, and i the outer loop.

In a situation like this, if the developer is confident that the rest of the task is trivial, the development of the pattern sequence can be stopped at this point and a placeholder used to indicate that there is more to be done to complete the sequence. In this situation we will use a placeholder called 'calculate values' as Pattern 10. However, if necessary, the process can be continued through `DECLARATION`, `ASSIGNMENT` and `EXPRESSION` patterns to find out how a dependent value can be produced by an expression. Calculating the values completes the solution of the initial problem because applying it leaves no smaller problem to be solved. If the nine patterns are then put together in Alexander's "pattern sequence" form, the following pattern sequence results.

Pattern 1 : `PROGRAM`.
Pattern 2 : `EXTERNAL RESOURCES`.
Pattern 3 : `FUNCTIONS`.
Pattern 4 : `MAIN`.
Pattern 5 : `STATEMENT SEQUENCE`.
Pattern 6 : `REPETITION`.
Pattern 7 : `LOOP`.
Pattern 8 :

```
┌─────────────────────────────────────────────┐
│INDEX LOOP                                   │
│ Pattern 9 :                                 │
│          ┌─────────────────────────────────┐│
│          │INDEX LOOP                       ││
│          │     Pattern 10 : Calculate Value││
│          └─────────────────────────────────┘│
└─────────────────────────────────────────────┘
```

This shows the structure of the solution as Alexander says it would - "in its sequence form it shows the user the process of unfolding, in sequence, in such a way as to allow a good building to be made, very easily, for the particular conditions of a given site." In our case it is not a 'building', of course, but the ‘nested loop’ structure of our solution, with the inner loop containing the number sequence generating pattern, is clearly evident in the diagram. Finally, the completion of the solution involves providing the code to fit the design, and the ‘code example’ sections in each pattern provide a guide to this.

7.5 Patterns and Teaching Material

The patterns used at this level are, more or less, the examples that would normally be used in a first programming course, packaged in the pattern format. The advantages of this approach are twofold. Firstly the pattern format generalises the example code by providing extra information about its use in different contexts. Secondly, it separates the pattern from the other teaching material that surrounds it. This separation can be optimised by reproducing the patterns in a separate document for ease of use in the assignment situation. In this way the pattern format addresses the two main problems that novices have in using the examples provided in their course material, finding them and using them. The patterns are written during the normal course of developing the other materials for the first programming course. They are software or design patterns in the sense of the patterns in the book “Design Patterns: Elements of Reusable Object-Oriented Software”(Gamma et al. 1995), and are only pedagogical in the sense that they are used in a teaching environment.

The advantage of adding patterns to the design process is threefold. First, it enables the tackling of coarser grained features in the original problem specification than would be possible otherwise (that is, there will be fewer steps in the decomposition process). Having identified a pattern that handles part of the problem you can effectively ignore that part of the problem and concentrate on the smaller problems exposed by the first pattern.

The second advantage is that the patterns discovered by this process will produce the ‘pattern sequence’ effect, which facilitates the design of the solution as shown by the evolving sequence of patterns through the successive iterations. The relationships between the patterns in a pattern language provide the connective power that enables sequences of patterns to solve problems larger than those solved by individual patterns in a pattern language. This is a case where the whole is more than the sum of its parts. The ‘pattern language’ enables large problems to be dealt with by building up a sequence of patterns.

What the network of relationships, illustrated in the pattern language, adds to the process, is an indication of the context for the next pattern. If you are currently ‘adding structure’ to `STATEMENT SEQUENCE`, for example, the pattern language tells you that there are only four patterns that you need to consider when deciding on the next pattern - all the others can be ignored. This focusing of attention on the relevant patterns is especially important in the case of novices.

The third advantage is the converse of the first. The power of the pattern is itself twofold. Firstly it enables the details of the part of the problem that it solves to be effectively ignored while the rest of the problem is tackled. Then, once all the patterns are identified and the overall design of the solution attained, it enables the details that were ignored in the earlier stages to now be filled in easily, because they too are part of the pattern. Thus a pattern consists of two main functions or forces. The outer shell, the name of the pattern, functions as a component in the process of building up an overall solution, while the material encapsulated in the pattern enables the component details to be filled in once the overall solution has emerged. In other words, time is not wasted on fine details until it is clear that the overall design does indeed solve the problem. Once the 'pattern sequence' is complete the code example in each pattern is used to write the coded solution.

But there are other advantages in terms of pedagogy due to the fact that the pattern language presented to the novice at any particular stage of her development can match match her current understanding. That is, the pattern language will evolve over time just as the novice's understanding does. This is made possible by the separation of the conceptual information required for design, from the implementation detail. Because of this partitioning all the patterns can be presented in a single document at the start of the course and the pattern language diagram relevant to the state of the current understanding of the students handed out at different times during the course. In this way the externalising of the thinking, discussed earlier (see Section 7.1), is actively promoted by the teaching material.

7.6 The Evolution of a Pattern Language

To illustrate how the development of a student's understanding can be represented by an evolving pattern language we show here how the first two program examples often used in a first programming course in Java can be presented in the form of a pattern language. The process of building a sequence of patterns that solves the problem from the language can be seen much more clearly. Usually the dynamic process of deriving the solution is only ever attempted in the lecture situation, but these live presentations are not available to the novice when they are needed most, and the static, lecture-note format does not capture the process at all well. We feel that the pattern language and pattern sequence shown here make the code examples less static, and more importantly show how new concepts can be 'attached' to the conceptual structure that the student already has.

Moreover, complex ideas made up of several smaller concepts, like the use of variables, form small pattern languages in their own right. Adding such complexes of concepts to a student's knowledge base, invariably involve a 'shuffling' of the concepts, a re-arrangement of the conceptual structure, in the student's head, and the following examples make this evolution explicit. In the following exposition the use of various Java constructs as the names of the patterns is simply a device to minimize the need for explanation. The patterns so named are both more than, and looser versions of, the Java concepts concerned. For example, the pattern called Class might be better named Class-as-Blueprint, Method might be better named Method-as-Action-Sequence, and so on.

Often the first code example presented to novices is the ubiquitous ”Hello, World!”, in which case the following pattern language applies (Figure 7.4):

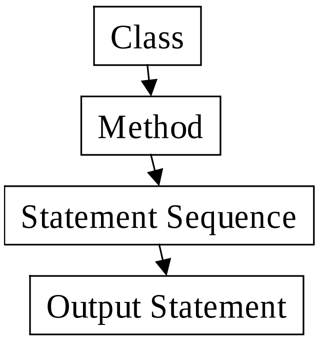


Figure 7.4. The Language for ”Hello, World!”

Notice that this pattern language can only generate one sequence. This is why this is the simplest possible program, as there are no real design choices to be made. The pattern language can develop only the one type of program, that being the output of a literal to the screen.

Class
Method
Statement Sequence
Output Statement

Figure 7.5. The Pattern Sequence for “Hello, World!”

As mentioned above, the next example often presented in teaching material is based on the variable concept, which, in isolation, can be regarded as a small pattern language in its own right, as shown in Figure 7.6:

Adding the new pattern language to the first results in the following structure shown in Figure 7.7 and we believe that this is important as a reflection of what must happen in the novice's mind. In the standard exposition, that is, without explicit pattern material, something like this ‘merging’ of concept structures must be occurring. By the use of example programs, which are, after all, made up of pattern sequences, the novice would be attempting to fit the isolated understanding of the variable concept into the understanding of the Java language that

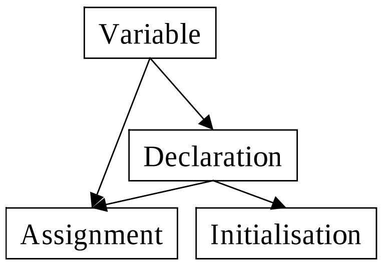


Figure 7.6. The Language for the Variable Concept

she already has. In other words, the novice merges the two pattern languages without any awareness that this is what she is doing. We discuss this merging of pattern languages in more detail in a paper published in 2005 (Porter et al. 2005). But our main contention here is that the explicit use of the pattern languages concerned helps make knowledge procedural rather than declarative.

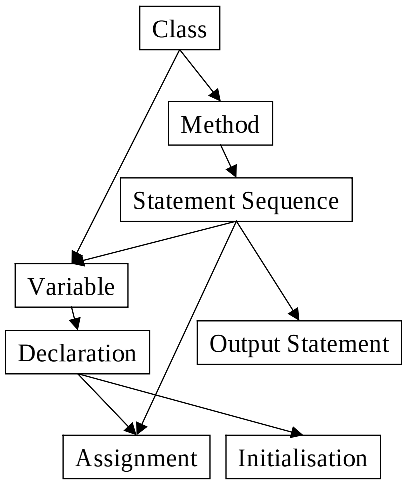


Figure 7.7. The Merged Language

One sequence, shown in Figure 7.8, derived from this language is based on the use of an instance variable. Another sequence, explaining the use and initialization of a local variable is illustrated in Figure 7.9, and a third, in Figure 7.10 demonstrates how an assignment statement can be used to change the value of an instance variable. We have not bothered to show the code examples here, but in the teaching material the code solutions would follow each sequence diagram so that the progression from pattern language through conceptual design, represented in the pattern sequence, to the code would be explicated completely.

Class
Variable
Declaration
Initialization
Class
Method
Statement Sequence
Output Statement

Figure 7.8. The Pattern Sequence for Using an Instance Variable

Class
Method
Statement Sequence
Variable
Declaration
Initialization
Statement Sequence
Output Statement

Figure 7.9. The Pattern Sequence for Using a Local Variable

The sequences shown here demonstrate the process of working through the pattern language discussed in Section 7.2 in a way that the code example on its own fails to do. Novices can follow the back-and-forth step-through procedure by which the program is designed in a way that is not possible with the bare code. So this is an important improvement on programming pedagogy but, more significantly it illustrates, for the teacher, how people progress through levels of understanding. New concepts, as they are learned, need to be fitted into an existing mental picture, and complex concepts usually cause a shuffling of the current understanding of the kind shown in this example. If nothing else, the current pedagogy, by failing to make this explicit, misses the opportunity to guide the development process. It has to happen, anyway, but letting it just happen by default means that developing the necessary *structure* is a hit-and-miss affair. The way that understanding develops over time can be represented in techniques like the merging of pattern languages as discussed in a paper published in 2005 (Porter et al. 2005).

7.7 The Pedagogy

So this is a powerful pedagogy on several levels. Each pattern is a small unit of knowledge, containing within it the means for the first four of Bloom's cognitive processes, knowledge, comprehension, application, and analysis (Bloom et al

Class
Variable
Declaration
Class
Method
Statement Sequence
Assignment
Statement Sequence
Output Statement

Figure 7.10. The Pattern Sequence for Using an Assignment Statement

1971). The pattern language structure adds weight to the fourth factor, analysis, and provides the means for the last two, synthesis and evaluation, through the contextual information and the creative process that it provides. Patterns lend themselves to the learning of a skill, like programming, because they provide the static knowledge plus the means of applying it, the dynamics of language. In fact patterns have been criticised for being just a teaching aid. But the fact that it can be said that "when designers become experts they discard patterns" (Mattson 1996) and that "patterns are limited because of what they are - a teaching tool"(Booch 1999) seems to miss the point that this is true of all expertise. Once a technique is integrated into an expert's practice its use occurs virtually unconsciously, like shifting gears in a car when driving (Skemp 1971, p. 55). So this is criticising a pattern for being a pattern - the written software pattern has become a mental pattern. Of course, this is the thrust of this paper; the correspondence between the specialised meaning of the word in the design pattern sense and the normal meaning of the word in the mental pattern sense gives the concept its educational power.

Bloom's categories	Learning to program
Knowledge	Tools, constructs, syntax
Comprehension	Relating concepts
Application	Flow, semantics
Analysis	Understand the problem
Synthesis	Create the solution
Evaluation	Assess other options

Table 7.1: Bloom's Categories in Pedagogical Terms. Adapted from (Bloom et al 1971)

But the pattern language idea has other advantages in the educational context. For example, the separation of the design idea (the pattern name in the language diagram) and the implementation details (the example section in the pattern form) helps to encourage designing the solution before coding. A second advantage is that the code examples used in teaching materials are representations of the final artefact not the process by which it is developed. Replacing them with a pattern language diagram containing the elements needed to develop the program followed by the particular pattern sequence is a much clearer representation of the program's development. So instead of just the Java code representing the use and initialization of a local variable, it would be preceded by the pattern language shown in Figure 7.7 and the sequence in Figure 7.9. This illustrates the *process* of developing the solution.

However the principle advantage is the correspondence between the pattern language concept and thinking. In particular, the way that understanding develops over time can be represented in techniques like the merging of pattern languages as discussed in a paper published in 2005 (Porter et al. 2005). The conceptual understanding that a novice has at any particular stage of his/her learning can be seen as a pattern language, and this language is modified as the novice learns new concepts. If a new concept is at all complex then it forms a small pattern language in its own right, and adding the new concept to the novice's existing language represents a merging of two languages, not just a simple attachment of a concept to a language, or of one language to another. People progress through levels of understanding, so new concepts, as they are learned, need to be fitted into an existing mental picture, and complex concepts usually cause a shuffling of the current understanding of the kind that can be represented as the merging of two pattern languages to form a new one. Pattern languages are thus useful at two levels of process. They drive the programming process in the manner discussed in the example above, but they also illuminate the learning process, and in particular the evolution of understanding over time.

As discussed in Chapter 5 the closest process in life to the learning of programming is probably the learning of the first spoken language. But there is a vast difference in how the process presents to the learner because the child acquires its first spoken language one word at a time, they are 'discovered patterns'. Meaning is the key to learning the use of a new concept and the critical factor for the learner in discovering meaning is context. The combinational aspect is also interesting. When a child speaks its first self-constructed sentence, its useable vocabulary, if 'useable' is defined in terms of what it is capable of putting into sentence form, is, most probably, only those words actually used in the first sentence.

So there is a vast chasm here, between the way that a programming language is learned and the acquisition of spoken language. Concepts in natural language are 'discovered' through use, not just presented in isolation as 'facts' to be learned. Language is acquired 'actively', not 'passively', it is learned by using it. The vocabulary expands 'naturally', via discovery through use. Acquisition and understanding are products of use, not rote learning. A pedagogy for programming based on the use of pattern language can reflect this evolution in a way that is simply not possible in the current methodology. This is how the disjunction between what a student *knows* about the programming language, and what she can actually put to use in solving a programming problem can be overcome.

Another advantage of a pattern language in the learning to program context is that this provides a mechanism for testing the efficacy of pattern languages generally. Already we have conducted a pilot study, reported in (Porter & Calder 2004), and made two attempts to test the use of pattern languages with novice programmers. The results of these experiments are discussed in Chapter 10.5 but the main point to emerge from the experiments is that basing them on the use of a pattern language is a way of making the subjective level of understanding apparent. There is a major difficulty involved in the attempt to measure empirically the performance of any particular skill caused by the fact that a skill of any complexity involves a mix of mental factors that are almost impossible to disentangle (see Section 8.4). As an externalisation of the thinking involved in designing a solution, the pattern language provides a mechanism of accessing the cognitive process involved, a view of the conceptual solution and its derivation expressed in the pattern sequence.

So, despite the difficulties we encountered in designing an experiment that addressed the problems of motivation and commitment, we feel that our pilot study which was based on interposing a test between the first and second programming courses and testing two groups of volunteers, one with exposure to a pattern language, and one, the control, without, did demonstrate the basic feasibility of measuring the effectiveness of different educational materials. The correlation between the understanding of the programming environment, indicated in our case by the results that students got in their first programming course, and their individual performance in the programming test did hold as expected, with the pattern-exposed students improving on their first course results more so than the members of the control group. Therefore given the correct experimental setup, an experiment based on the pilot study should clearly indicate any difference in the effectiveness of the materials being used.

So the pattern process explored in this chapter derives its effectiveness from the fact that it is based on the way that the human mind works. Concepts are not hard-wired into our thinking, they are patterns, not the rigid definitions that a system of logic requires. The best illustration of the difference between a rigidly formal system and an informal one based on patterns is probably the use of numbers. In the informal usage employed by humans, it matters not a whit that there are two number formats, integer and decimal, we mix them in our thinking freely because number is a pattern. We just divide five by two and let context decide which of the two forms matters in each case. However, such ‘looseness’ is not permissible in a mechanical system, here the two forms must be explicitly separated because calculations using the different forms must be performed by different logical circuits. Patterns are the basic grist of our psychology, so here we need to explore the psychology of programming.

Chapter 8

The Psychology of Programming

Programming is never going to be easy, since it forces otherwise sane people to think like computers.

(Stephan Somogyi)

He only earns his freedom and his life
Who takes them every day by storm.

Johann Wolfgang von Goethe

8.1 What is Programming?

Ostensibly, programming is nothing more than instructing a machine, giving it a list of precisely determined operations to carry out. This would appear to be easy, even trivial, yet history shows that this is definitely not so. It is a prominent feature of the history of computing that programming is seen as a difficult task, even in cognitive terms alone. There have been three basic responses to the perception of this difficulty. One response has been to modify the context, considered in narrow terms, in which programming takes place, that is, the notation of the means of communication between programmer and the computer itself, the development of programming languages approach. Another approach has been directed at changing the context, in much wider terms, the software engineering approach. And a third has been to examine particular aspects of the overall task in terms of the particular cognitive issues that each one raises, the cognitive fitness approach.

The first response seems probably to founder on the fact that, ultimately, any 'programming language' is constrained to strict logical form by the fact that it must ultimately be translated into machine code. One's thinking about a programming problem is therefore driven along rigid logical lines even though the language is made to appear as close to natural language as possible. A major difficulty with the second response is that, although it is soundly based in its concentration on the non-coding aspects of programming, it seems to carry across the impulse to formalise them as well, so that it tries to escape from one formal system by instituting a system of design that is hardly less formal than the one it is trying to escape. Both these responses miss the point that solving problems is an informal mental process not a formal or algorithmic one. A formal problem solving process is one based on searching a set of possible solutions for the correct one (Newell, Shaw & Simon 1963, p. 113), but it is simply not the case that every problem has an existing set of possible solutions and, in this situation, finding one is a matter of *creation* not *search and test.* This leads us to the third approach, fitting the task of programming to the way that the mind works. Of course, by implication, this requires a theory of mind.

The belief that programming is difficult begs the fundamental question - why is it so? Considered at the coding level, the actual means of interaction between programmer and machine, the task seems to be not that complex at all. After all the computer has very few native actions that it can perform, its operating repertoire is quite limited. It would seem, therefore, that the difficulties that programming presents are due more to the nature of the operations, or at least, to the way that they are presented to the programmer, than to their number, and it is this factor that lies behind the continual push for 'programming languages' that are 'easier to use'. But while there is undoubtedly some truth in this approach it seems to us that the difficulties are related to what we try to do with the operations available. That is, considered as a language, this is one with which we try to write extremely elaborate stories with a very small vocabulary. It is what we are trying to do with our limited 'language', write sophisticated programs, that introduces the complexity - we have complex ends not complex means. So the difficulties result in producing the "design metascript", a high level schematic representation that drives the design process which provides the basis of experienced programmers' expertise (Adelson et al 1984, p. 473). We must come to "realize that what we are called to design [has become] so sophisticated, that Elegance is no longer a luxury, but a matter of life and death. It is in this light that we appreciate the view of programming as a practical exercise in the effective exploitation of one's powers of abstraction" (Dijkstra 1982, p. 48) rather than, one might add, the manipulation of a formal logic.

> Although the program made by the programmer is his final product, the computations evoked by it are the true subject matter of his trade: he has to guarantee that the computations - the 'making' of which he leaves to the machine - evoked by his program will have the desired effect. As a result, he has the duty to structure his program in a useful way, where usefulness (among other things) implies that the form of the program admits trustworthy statements about the corresponding computations. The second theme is that the mental aids available to the human programmer are, in fact, very few. They

> are enumeration, mathematical induction and abstraction, where the appeal to enumeration has to satisfy the severe boundary condition that the number of cases to he considered separately should he very, very small. The introduction of suitable abstractions is our only mental aid to reduce the appeal to enumeration, to organize and master complexity. Mathematical induction has been mentioned explicitly because it is the appropriate (and only!) established pattern of reasoning by which we can understand programs with either repetitive clauses or recursive procedures.
>
> (Dijkstra 1982, pp. 1-2)

Djikstra then works though the process of developing two separate programs and goes on to point out why he thinks the difference between the two solutions - the care with which the decision as to where the interface between the successive levels of each should be put - is less significant than the similarities between them.

> Personally I am much more impressed by the similarity of the ways in which the two rather different programs have been constructed. The successive versions appear as successive levels of elaboration. It is apparently essential for each level to make a clear separation between "what it does" and "how it works". The description of "what it does", the definition of its nett effect, requires introduction of the adequate concepts, and both examples seem to show a way in which we can use our power of abstraction to reduce the appeal to be made upon enumeration.
>
> (Dijkstra 1982, p. 13)

It is interesting that Djikstra here mentions that the differences between the two programs are *less* significant that the similarities in the way they have been constructed. This points to the thrust of the pattern idea, that solutions have common ground in their underlying concepts in the 'problem space' - "what it does" rather than "how it works", that is repeating form rather than logical shape.

Whenever there is a failure in "the effective exploitation of one's powers of abstraction" it is a result of a deficiency in theory, we simply don't yet have the correct view of the situation in which we are working. This has been a constant state of affairs in software development, because the pace of change, driven partly by the rapid advance of the electronics that underlies it and partly by the drive to computerise ever more complex tasks, has been so great. On the face of it, the task of the software developer seems so much simpler than that of the hardware designer, but the difference is that the later is always working within long established theory. The main problem faced here is the practical difficulties of dealing with the 'fuzziness of analog nature', of staying 'within tolerance', not a lack of theory.

> For the understanding of his source components the hardware designer

> has as a last resort always physics and electronics to fall back upon: for the understanding of his target problem and the design of algorithms solving it the software designer finds the appropriate theory more often lacking than not. How crippling the absence of an adequate theory can be has, however, only been discovered slowly.
> With the first machine applications, which were scientific/technical, there were no such difficulties: the problem to be solved was scientifically perfectly understood and the numerical mathematics was available to provide the algorithms and their justification. The additional coding to be done, such as for the conversion between decimal and binary number systems and for program loaders, was so trivial that common sense sufficed.
> Since then we have seen, again and again, that for lack of appropriate theory, problems were tackled with common sense, while common sense turned out to be insufficient. The first compilers were made in the fifties without any decent theory for language definition, for parsing, etc., and they were full of bugs. Parsing theory and the like came later. The first operating systems were made without proper understanding of synchronization, of deadlock, danger of starvation, etc., and they suffered from the defects that in hindsight were predictable. The indispensable theory, again, came later.
> That people have to discover by trying that for some problems common sense alone is not a sufficient mental tool, is understandable. The problem is that by the time the necessary theory has been developed, the pre-scientific, intuitive approach has already established itself and, in spite of its patent insufficiency, is harder to eradicate than one would like to think.
>
> (Dijkstra 1982, p. 340-1)

8.2 Theory and Practice

This is, of course, the difference between art and science. Some of science is art in the sense that one is working ahead of, and therefore, in the absence of theory. But most of scientific endeavor is really technological in scope, one is working within the limits of theory. Art, on the other hand, is *never* just working within the limits of theory, it is always, to some extent, pushing the bounds of understanding. It isn't art if this isn't so. In some sense this is an expression of the difference between the 'real' world and the 'virtual' world of representation. The 'real' world is a continuum, Kant's "complete community" of substance (Kant 1881, p. 184), it is fundamentally analog in nature, at least at the level at which we, as humans, apprehend it. But the 'real' world has to be represented in symbolic form in our minds and symbolic form is *always* discrete, *never* analog. A lot of what is apparently paradoxical in nature is caused by this disjunction between reality and its representation - Zeno's Paradox is only the most obvious example - and leads to von Neumann's contention that although, in principle, a measuring device can be described in terms of universal natural laws, doing so requires the loss of its measurement function (Pattee 2001*a*, p. 349). It is probably something of this sort that underlies the apparent paradoxical duality of the nature of light, and, indeed Descartes' notion of the dual nature of brain-mind - the brain exists in the 'real' world, the mind in the world of 'ideas'.

It may therefore be that programming is destined to be always working ahead of theory in Dijkstra's terms, that in this field we need as much of artistic intuitive feeling as we do of basic knowledge - this is an art. As with music, theory cannot drive the composition of the artefact in the same way that scientific theory does a technological project. In other words it can't be formalised. So the problem seems basically to be that when we are programming we are "forcing our interactions into the narrow mold provided by a limited formalized domain" (Winograd & Flores 1987, p. 75) which stifles the source of our creativity, the power of abstraction and metaphor. The programming domain provides the *tactical means*, the operators, to solve problems. What it does not, and cannot, provide is the insight behind the task of combining them to solve a particular problem, the *strategy*. "That strategic knowledge can be considered separate from knowledge of operators is shown by the fact that a solver may know *how* to apply either of two operators in a given situation but still not know *which* to apply" [emphasis in original] (Lewis 1981, p. 87). The difference between a competent problem solver and a novice is exactly the presence or absence of *strategic* level thinking. Programming is an *activity* of the mind, not just a collection of static facts, and it is important, perforce, to have some appreciation of what might be called the psychology of programming. In fact it can be seen that most of the developments in programming languages and methodologies have been explicitly predicated on the belief that they will improve the performance of programmers by addressing factors that are essentially psychological in nature (Sheil 1981, p. 101).

> Although such claims are usually advanced informally, there is a growing body of research which attempts to verify them by controlled observation of programmers' behavior. Surprisingly, these studies have found few clear effects of changes in either programming notation or practice. Less surprisingly, the computing community has paid relatively little attention to these results.
>
> (Sheil 1981, p. 101)

It has been said of this research that it is based on "an unsophisticated experimental technique and a shallow view of the nature of programming skill" (Sheil 1981, p. 101). However this should not be surprising. We suggest that it is the fate of any attempt to measure the level of creative skill of any kind to fall into these particular traps, it is simply the nature of the beast. By definition, creativity implies an element of the unexpected, which is the complete antithesis of the science of measurement. Measuring always involves a scale, a proportional system, and this requires some expectation about the relationships being measured. So, for example, in the debate over structured programming in the sixties and seventies, attempts were made to measure the difference in programmer comprehension promoted by different programming notations (Shneiderman 1976*a*). In essence, this amounts to positing a relatively fixed relationship between syntax and semantics, a claim that syntactic form is mostly about meaning. While this is superficially true, it must be remembered that the real relationship is convention, and this is an arbitrary connection, not one based on any sort of universal principle. Testing on this basis is measuring the degree to which conventional norms of expression have been absorbed, so it is not surprising that the experimenters found that, for example, while the logical IF is significantly easier than FORTRAN arithmetic for novices, this advantage dissipates with experience (Shneiderman 1976*a*). This result supported earlier findings that had suggested that experience with particular linguistic form is more significant than the actual form itself (Sime, Green & Guest 1973).

The demand that "the methodological recommendations of computer science should be recognized as empirically testable, psychological hypotheses" (Sheil 1981, p. 101) is both understandable and laudable but probably unrealistic. It is understandable because, as Gannon and Horning point out, it is not possible to logically prove the principles that underlie projected changes (Gannon & Horning 1975, p. 10), but not only is it based on the mistaken belief that science is *simple* methodological empiricism, it overlooks the fact that programming, in particular, is an essentially creative activity. The practice of science is *not* simply the application of the formal rules of logic and mathematics to experience (West 1997). These are important tools in understanding the world, but understanding is the process of building meaning, and, as such it is fundamentally creative rather than formal in spirit. "Scientists and engineers were plying their disciplines with imagination long before their fields were formalized and recognized" (Fabian 1990). Progress in science depends almost as much on imagination as does art.

> Our conceptual imagination, like its artistic counterpart, draws inspiration from contacts with experience. And like the works of imaginative art, the constructions of mathematics will tend therefore to disclose those hidden principles of the experienced world of which some scattered traces had first stimulated the imaginative process by which these constructions were conceived.
> When experienced orderliness is taken to be an embodiment of geometry, it may become possible to test its correspondence to experience. The observation of relativistic phenomena has served as an experimental test for deciding whether the material universe was an instance of Riemann's geometry formulated in space-time by Einstein's rules, when combined with the assumption of trajectories being geodetics.
> (Polanyi 1958, p. 46)

But the empirical verification followed the original insight, and, as Einstein himself said, although "nobody who really goes into the matter will deny that the world of perceptions determines the theoretical system in a virtually unambiguous manner, ... no *logical* [emphasis added] way leads to the principles of the theory" (Einstein quoted in (Weyl) 1959, p. 153). This points to the fundamental correspondence between all the nominally different types of thinking - the child learning to understand the world and the scientist searching for meaningful explanations are both engaging in the cognitive process known as "inductive discovery" (Greenfield 1984, pp. 27–28) not just pure logical deduction. "The process of making observations, formulating hypotheses and figuring out the rules governing the behavior of a dynamic representation is basically the cognitive process of inductive discovery ... the thought process behind scientific thinking" (Greenfield quoted in (Prensky) 2001, p. 45). Science is a way of thinking about the world, an analytic way of thinking, to be sure, but, as Kant realized, analytic reasoning can't tell you anything that isn't ultimately self-evident. Synthesis is just as important as analysis, indeed you can't have one without the other - "thoughts without content are empty and intuitions without concepts are blind" (Kant 1881, p. 45).

> One of the most vital abilities of a software designer faced with a new task is the ability to judge whether existing theory and common sense will suffice, or whether a new intellectual discipline of some sort needs to be developed first. In the latter case it is essential not to embark upon coding before that necessary piece of theory is there. Think first!
>
> (Dijkstra 1982, p. 341)

The trouble is that the expert has the theoretical insight so deeply embedded in her thinking that it is easy to forget that it is there. She doesn't have to "think first", most of the time the integration of the theory in her thought patterns is sufficient for the task at hand, the analysis is mostly already accomplished. This points to a major disjunction between the way that we teach programming and the way that experts actually program. Given the task of formulating a procedure to find the largest value in set of N positive numbers, nearly all experienced programmers will immediately produce code to this pattern:

```
int largest = 0;
for(int i = 1; i <= N; i++) {
        int temp = <get the ith value>;
        if(temp > largest) {
                largest = temp;
        }
}
```

> Consider how that procedure could have been generated or understood. One way, the way that you might believe it was done if you

> read the literature on programming that is written by computer scientists, is by formulating a loop invariant. For this loop the appropriate invariant is that, at the end of the kth pass through the loop, m $\geq$ a $_{i}$; for j $\epsilon[1, k]$. Once formulated, that invariant can be proved by induction. Having been proved by induction, it can be instantiated for k = N and now constitutes a proof that m is a maximum of the set, which is the desired result.
> The problem is that nobody does it that way. And the reason nobody does it that way, except in introductory programming courses, is that it takes too long and is far too complicated. If you know how to program, you would neither generate this program nor synthesize an understanding of it. You would know the answer. You would recognize the problem, key directly into that knowledge, and pull out a working procedure.
>
> (Sheil 1981, p. 117)

The disjoint here is that the experienced programmer *knows* the pattern and simply produces the solution in response to the problem without having to think about it. As Sheil points out “the compelling subjective evidence for this ... is the complete absence of any introspective trace whatsoever” in the experienced programmer’s mind (Sheil 1981, p. 117). But, as educators, we have to explain the solution to students and the fact that our solution is written in programming language terms leads us into producing dubious explanations based on the code (the implementation ‘space’) rather than the concepts (the conceptual ‘space’). We stay in formal logic mode induced by the programming language even though we did not, ourselves, explicitly produce the solution in that way. The usual textbook explanations are appropriate to people used to thinking in terms of logical formalisms, hardly a mode of thinking we would expect of novice programmers. As experienced programmers we have a concept, a *pattern* in Alexander’s sense in fact, that fits the task so we simply apply it, but as educators we are aware that the novice does not have the concept, so what we need to do is to reproduce the process by which we acquired the concept.

But this is where the difficulty arises because the process of concept acquisition is largely unconscious, a matter of having solved the problem several times. Of course the novice *does* need to have some appreciation of the logic behind the solution but explaining it in those terms might not be the best way, or the most honest, of helping her to acquire it. Logic, by definition, is a controlled progression, one step follows automatically, predictably in other words, from the previous one. Creativity is, also by definition, the exact and complete opposite - an unpredictable, and therefore, uncontrolled progression, an operation involving many fits, starts and retracings.

> The mere fact that creative mental processes are *mental* processes does not ensure that they have explanations in the language of psychology under *any* of their descriptions. It may be that good ideas (some,

> many, or all of them) are species of mental states which don't have mental causes. Since nothing at all is known about such matters, I see no reason to dismiss the intuitions that creative people have about the ways in which they get themselves to act creatively. The anecdotes are, I think, remarkably consistent on this point. People with hard problems to solve often don't go about solving them by any systematic intellectual means (or, at least, if they do they often aren't conscious of the fact they are doing it). Rather they seek to manipulate the *causal* situation in hopes that the manipulated causes will lead to good effects.
>
> The ways that people do this are notoriously idiosyncratic. Some go for walks. Some line up their pencils and stare into the middle distance. Some go to bed. Coleridge and De Quincy smoked opium. Hardy went to cricket matches. Balzac put his nightgown on. Proust sat himself in a cork-lined room and contemplated antique hats. Heaven knows what De Sade did. It's possible, of course, that all such behaviours are merely superstitious. But it's surely equally possible that they are not. Nothing principled precludes the chance that highly valued mental states are sometimes the effects of (literally) nonrational causes. Cognitive psychology could have nothing to say about the etiology of such states since what it talks about is at most ... mental states that have mental causes. It might be that we are laboring in quite a small vineyard, for all that we can't now make out its borders.
>
> So far I've been concerned with cases where mental states aren't (or, anyhow, may not be) contingent upon mental causes. The point has been that the etiology of such states falls, by definition, outside of the domain of the explanatory mechanisms that cognitive psychologists employ; cognitive psychology is about how rationality is constructed, viz., how mental states are contingent on each other.
>
> But, in fact, the situation may be worse than this. Cognitive explanation requires not only causally interrelated mental states, but also mental states whose causal relations respect the semantic relations that hold between formulae in the internal representational system. The present point is that there may well be mental states whose etiology is precluded from cognitive explanation because they are related to their causes in ways that satisfy the first condition but do not satisfy the second.
>
> (Fodor 1975, p. 202)

All of which justifies the role of philosophy in the intellectual project. The context of cognitive activity is the human condition, and the context of the human condition is reality, so it is perfectly understandable that not everything that is cognitive in function will be cognitive in origin, and that not everything in the human condition will be explicable in purely human terms. If there is an overarching thrust involved in the idea of pattern language it is that form is the product of context (Alexander 1964, pp. 15-6). But given the universality of the universe, that is, everything that we know is ultimately an effect of the existence of the universe, decomposition into "special sciences"[1] is useful and necessary, but somehow the whole must be preserved. If it is anything, philosophy is the attempt to put the pieces back together again, to counter the tendency to reductionism, to provide context for all our varied explorations of nature.

> Philosophy studies the fundamental nature of existence, of man, and of man's relationship to existence. In the realm of cognition, the special sciences are the trees, but philosophy is the soil which makes the forest possible.
>
> (Rand 1982, p. 2)

But all this means that there is an often overlooked aspect of 'myth' in creativity. The point is that we need myth to be functioning individuals because we can never know everything about any given situation or field - subjective knowledge is always partial knowledge. Unfortunately the word 'myth' has acquired the connotation of 'unfactual', or even 'lie', when in its original sense it was deliberately neutral to the idea of 'objective truth' in any metaphysical sense. Myth was a way of defining or explaining what might loosely be called a universal 'human' truth, a factor of human existence that is always true, such as, for example, that exemplified by the "choice of Hercules" story, that life requires making choices about one's personal place in the totality of existence. As such, myth is a story that illuminates some aspect of life regardless of the actual truth of every detail expressed, its veracity is not the point of the story. In this sense it is like the modern novel or film that is known to be based on 'actual events', but which makes no claim on the 'absolute' veracity of every detail, the narrative or even entertainment value overrides literal truth.

Myth, then, is just a fact of life because it is never possible to be completely sure of every detail in a given situation, this is simply a given of human existence, so to have any form of explanatory or defining text involves some degree of 'myth' in the original sense of a 'defining story'. The purpose of myth is to illuminate life, to point to the 'universals' behind the shadows in Plato's sense. To demand absolute correctness in every detail is not only silly it is impossible - we want epistemic truth not metaphysical absolutism. There is always a devil in detail and that evil is the potential to overwhelm understanding. Myth extemporates the fundamentals and thereby is a form of abstraction, maybe even the *original* form.

To this extent, then, what an educator always does is to construct a myth, so the choice is not between myth or truth, but which myth to use. This is not

[1]Fodor uses the term to denote sciences that are not physics, on the basis that physics is the most general description of reality, not on the positivist basis that ultimately, "all true theories in the special sciences should reduce to physical theories (Fodor 1975, p. 9).

to make any judgement about the existence of otherwise of an absolute truth, simply to assert that because we cannot cognitively deal with every everything simultaneously we *need* to simplify in order to understand. The implication is not merely that we cannot know all there is to know, but that what we know is only partially true (Clark 1990, p. 146).

This mythmaking aspect is also true of programming. It is hardly likely that a program of any complexity is produced by thinking entirely in code. The programmer simply must have thought about the solution in general terms and this form of the solution clearly bears a mythical relationship to the coded one, it is 'defining form' rather than 'implemented form'. So what is it about the myth form that enables it to illuminate the fundamentals? Mostly it would would seem to be the power of abstraction. Just as myth lifts the essential message about life that it is presenting out of the warp and weft of everyday life, so the general thinking separates the solution principle from the coded solution.

Of course it is this very same abstractive power of metaphor that has driven the change in meaning of the word, 'myth', from 'defining story' to 'untruth'. Abstraction implies the taking of an idea, a form, out of its original context and this is also the force of the idea of deception. The Greek word, $\delta\iota\alpha\beta\alpha\lambda\lambda\epsilon\iota\upsilon$, to deceive, has the original literary meaning of 'to throw across', that is, 'to take something out of its original context and put it in another with the intention of making it appear that it belongs there'. (The phrase 'red herring' would appear to be a modern manifestation of the 'throwing across' metaphor.) We alluded to this power of metaphor to deceive, or, at least to *mis*lead, when we talked about the 'brain as computer' metaphor (see Section 3.2). Furthermore, the Greek word for 'deceiver' or 'one who throws across' is rendered 'diabolus' in Latin and comes down to us as 'diabolical'. Hence, also, the saying that 'the devil[2] lives in the detail', pointing out, perhaps, the tendency of a mass of detail to obscure meaning, to deceive. All this points to a fundamental fact about meaning. Context is everything in meaning, so much so that, both too much and too little context in presenting a concept can cause semantic problems. Trying to encompass context in all its gory detail can deceive one, but, equally, abstraction (taking out of context) can also be deceptive.

What we want, then, in dealing with complexity, is abstraction without the abstractive act distorting the essential meaning. The pattern idea avoids these contradictory tendencies in abstraction by two means. Firstly, the pattern abstracts the essential features of using the concept it encapsulates by identifying it, not in purely conceptual or theoretical terms, but from *actual practice*. That is, patterns are *discovered* not made[3]. Proceeding from actual usage means that the features of the context that are essential are not lost in the act of abstraction - in

[2]Diabolus is the root for the word 'devil' in many languages including English.

[3]One needs to be a little careful here because this emphasis on the fact that patterns occur in practice does not mean that there is no theoretical component in their 'discovery'. One still has to spot the underlying conceptual connections in the practical form and this often involves theoretical and even logical types of thought. "For example, the pattern PARALLEL STREETS

a sense the pattern discoverer is mining domain practice, not domain theory. Secondly, the pattern is always presented as one item in a language precisely because this sets it in context in terms of usage - the arrow from one pattern to another, says that applying the first has set up the context, imported that detail from the concept's environment that is necessary for it to retain its essential meaning, for the second. Pattern is 'metaphor with context' or 'abstraction with context', phrases which are *almost* a contradiction in terms. What saves them from contradiction is the fact of a pattern's place in a language, the detail-importing or context-setting power of the language set out in the pattern language diagram.

The critical interdependence of symbol and the system in which it occurs is the factor that is most often overlooked in various definitions of 'information', and which causes most of the confusion that surrounds 'information theory'. This is the problem of meaning, again. Any attempt to define information independent of semantic value (fitness to the dynamics of its context), that is, in totally abstract terms, is bound to give rise to misleading explanations. So, for example, many attempts to relate entropy to evolution seem to falter because they are based on a semantics-free definition of information (Pattee 2001*a*, p. 347). The molecules that act as symbols in the genetic coding system are only symbolic in terms of the *functioning* of the genetic coding system as a whole. "A molecule becomes a message only in the context of a larger system of physical constraints which I have called a 'language"' (Pattee 1969, p. 8) that derives, ultimately, from the semantics of the organism's relationship with its environment, because without this relationship there would be no 'information' to encode. In a sense it is the treatment of the genetic code as a set of symbols in the abstract, that causes the confusion, just as the programming language's degree of abstraction does.

One of the main reasons for our inability to measure, in any sensible fashion, the quality of a program derives from this obscurity of conceptual structure in the code itself. Many of the 'stylistic' additions to coding practice, such as the structured programming and literate programming movements were attempts aimed precisely at addressing the lack of conceptual structure in code. There are only two ways to assess the 'quality' of a program using only the code and they are to run the program or to understand it. The first method limits the assessment of quality to the bare fact that it works in all the circumstances that the tester can envision, but, as this is much easier than the second method, quality assessments often default to it. Building up an understanding of a program from the bare

was discovered by purely mathematical reasoning, based on the forces which connect high speed vehicular movement to the needs of pedestrians, the problem of accidents, the huge travel time, the very slow average speeds, etc. At the time we discovered it, we were unaware that it actually was an emerging pattern in the world of the 1960s and only later realized that separated parallel arteries, without cross streets, was emerging as a pattern in several major cities. In this same sense, it was possible to 'discover' uranium, by postulating the existence of a chemical element with certain properties, before it had actually been observed" (Alexander 1979, pp. 259-60). The fact that these patterns occurred in practice and were identified as patterns virtually independently follows Kant's dictum that though all our knowledge begins with experience, it does not follow that it all arises out of experience (Kant 1881, p. 1).

code is a difficult and time consuming process because of the obscuring effect of the sheer mass of detail. Intuitively, imposing some form of structure on the bare code is a way of making it 'easier' to read, and therefore, to understand, and studies with "literate programming" (Knuth 1984) scripts compared to conventional structured style programs do indicate that this style made modifying them easier for novice programmers, indicating an improved level of understanding (Bertholf & Scholtz 1993).

But it is, obviously, easy to exaggerate this effect - 'stylistic' structure is *not* the same thing as conceptual structure - and in any case, stylistic structure can only assist conceptual understanding in terms of an already written program, or part thereof. While program comprehension is clearly related to programming skill it is more about other aspects of software engineering (Brooks 1978), such as code maintenance, than those involved in the design stage (Young 1996). Nevertheless understanding code written by others is an activity that is recommended for novices, and this is studied in Brooks, "Towards a theory of the comprehension of computer programs" (Brooks 1983) and in Wiedenbeck, "The initial stage of comprehension" (Wiedenbeck 1991). Moreover, program comprehension is another one of those activities that relates closely to the pattern idea of reusing expertise which was one of the main motivations behind the introduction of Object Oriented programming. Because OO was believed to be a better environment in which to model real world entities and activities this was expected to make reusing concepts, and therefore code fragments easier. However, a comparative study of program comprehension in novices in the Object Oriented and procedural paradigms, concluded that the trade off between the increased complexity of the programming language and the intuitive fit of the Object Oriented paradigm with the real world, was not entirely clear cut (Ramalingam & Weidenbeck 1997). And it is here, we feel, that pattern languages fit in by structuring the elements of OO thinking in terms of process.

In essence, the pattern language idea is a way of abstracting, of capturing the 'spirit' of a code construct safely, that is, without distortion - the pattern carries its context with it, or, at least, that part of its context that it needs to make sense *in terms of reusing it.* This contextual feature is possibly the most important aspect of the pattern idea in avoiding the common problems that a high degree of expertise in a particular field causes, because most of the 'danger' of science derives also from the power of abstraction to distort. Being an expert in her domain means that the scientist carries around most of the context in which her work exists unconsciously - the sense of process is implicit. If she is conscientious about retaining this awareness, then it is less of a problem. however, not only can we not rely on every scientist being conscientious in this way, it does not address the problem of communicating her ideas to people who by the nature of their being non-expert in her field, do not have the same 'unconscious' contextual background - sense of process - for the ideas being presented. The words, the symbols, are perceptually the same for everyone, but meaning is contingent on experience, it derives from context.

8.3 The Problem of Meaning

As with any intelligent activity, the key to programming is understanding, and the source of understanding is 'meaning' - a concept is only a concept insofar as it 'means' something, that it 'acts' in terms of my life. A bird can be trained, 'programmed' in fact, to utter phrases like "Polly wants a cracker" but to regard this as symbolic behaviour is missing the point that for a word to be a symbol it must represent a concept, it must carry meaning. This leads to a fundamental question - what is 'meaning' in terms of the physical reality of the world? Like Karl Pearson's famous question - if life is entirely a physical process then what is it that physically distinguishes the living from the lifeless? - this points to one of those fundamental disjoints that cause us to resort to dualist explanations (Pattee 2001*b*). It is clear that in responding to physical reality we develop a symbolic model that is the entire basis for the way that we act in the world, and it is this symbolic representation that carries meaning, that relates the Platonic form to the real. Meaning is therefore some sort of relationship between life (biology) and the physical reality in which it occurs - meaning is an empirical, not a definitional relationship and this can clearly be seen in those cross-domain mappings in the conceptual system that we call metaphors. " In short, the locus of metaphor is not in language at all, but in the way we conceptualize one mental domain in terms of another" (Lakoff 1993).

But to express meaning we must use symbolic form, and, as Max Born pointed out, they are many unanswered questions about the relationship between physical laws and the mathematical symbols on which their representation depends. If, as Born saw, all experience, "everything without exception", is entirely subjective, then it is only by using symbols that we can express any objective sense of our subjective, private experiences (Max Born cited in (Pattee) 2001*a*, p. 341). He proposed experiment as the condition for the objective use of symbols, a role he called "decidability", so that " if a symbolic expression lacks empirical decidability potentially available to all observers it has no necessary relation to any objective reality" (Pattee 2001*b*). But even if a particular representation does bear an empirical relationship to its basis in reality, that is, it provides an empirically decidable description, it may still fail to carry any meaning. As Einstein once said, everything has a correct scientific definition, but it might be "a picture with inadequate means, just as if a Beethoven symphony were presented as a graph of air pressure" (quoted in (Pattee) 2001*a*, p. 344).

This touches the related problem of reductionism. It seems to be the case that some aspects of higher level forms are impossible to represent at the next level down. For example, all snowflakes are individual, their shape reflects a unique history *as well as* the symmetry derived from the molecular level. So the break between elemental and aggregate form is an epistemic disjunction, to adequately understand the whole snowflake complementary descriptions are required. This is the 'measurement problem' again, the fundamental difference between discrete entities (elemental form) and the continuous nature of experience (experiential form).

> Solutions that are the most stable (stationary states) lead to the atoms and molecules that are the basic forms within the energy domain of living systems (I omit the lower level fundamental particles from this discussion). They are elemental in the sense that one atom or molecule of a given form is as good as another in building higher level structures. In other words, the stability of these forms is such that their local environmental history is irrelevant and can play no role in their description. It is also at this level where complementary descriptions of the wave/particle duality are an empirical necessity. ... At the next level are forms made up of elemental forms. They are more complex but less stable. This is the level of macro-molecules and crystals that may have a timeless overall symmetry, but that also show innumerable detailed shapes that depend on their individual local environmental histories. ... Aggregate forms have a lawful elementary description, in principle, but their complete structure is not reducible to the laws of elementary forms because their individual environmental history is not in the language of the elementary forms.
>
> (Pattee 2001*a*, p. 344)

Incomputability is another example of an epistemic break. It simply means that a situation is not definable within the terms of the defining formalism. A program fits this category, it is a solution 'reduced' to a description in the limited means provided by the programming system, so it may even be perfectly correct in those terms (that is, it executes) yet fail to address the real world problem it was supposed to solve. So 'meaning' is a complicated relationship, it would seem that meaning is only possible in the context of a web of semantics. But even within the linguistic system itself, meaning is not as straightforward as we like to think. This is best considered using an example and for this purpose I have chosen the word 'file' in the context of 'information'. Logically the best place to start an examination of the 'file' idea is with a dictionary definition and see where that takes us initially. The online version of the Oxford English Dictionary gives us, among other items, the following list:

> A string or wire, on which papers and documents are strung for preservation and reference. In recent use extended to various other appliances for holding papers so that they can be easily referred to.
> A catalogue, list, roll.
> A collection of papers placed on a file, or merely arranged in order of date or subject for ready reference.
> A collection of related records stored for use by a computer and able

> to be processed by it.
>
> (Oxford Dictionary, 2005)

Some other definitions that might throw some light on our investigation are:

> A row of persons, animals, or things placed one behind the other. the common file = the common herd (obs. or arch.) in file: one after another, in succession.
> *Chess.* One of the eight lines of squares extending across the board from player to player.
>
> (Oxford Dictionary, 2005)

Even from these 'defining' ideas it is clear that the concept of 'file' is a very 'slippery' one. Perhaps, before the advent of the electronic age, the idea that the concept of 'file' was carried by the physical apparatus that kept papers together could be countenanced, but even here we are on shaky ground. Does said physical apparatus still carry the concept, the same meaning, when it is empty of papers? It is still the identical physically existing object that it was when it contained the papers, but most of us would probably be uncomfortable with calling it a file. And what if all it contained was several blank pieces of paper - would we still want to refer to it as a file? A file, at least in the sense that we are using the word here, containing no information is a rather unintuitive idea, at least to our modern minds. Maybe for people in the earlier age where the concept was attached to the linking apparatus, and to the physical placement one after the other implied by the two extra definitions, this was acceptable. This is one of the points about 'meaning' - it is affected by context, and the current 'age' is part of the context for a word as meaning can change over time. What you and I understand about the word, 'watch', as in personal timepiece, is quite different from what a person in 1910, before the widespread advent of wrist watches, would have.

The attempt to 'locate' the concept in a material manifestation of some kind in physical space is clearly fraught even when we restrict our thinking to earlier times. Maybe the second item in the Oxford's primary list above can ground us more securely - "a catalogue, list, roll". Clearly these ideas are also rather ambiguous and unclear. How much do these concepts themselves depend on some physical material manifestation? Do we consider an organised collection of items, of, say, the symbols for goods to be bought when I go to the supermarket, a 'list', even when it 'exists' solely in my memory? And, of course, even in trying to discuss the original concept one can't help introducing other ideas such as 'organised collection' that are themselves pretty hard to pin down.

The third item is, more or less, a variation on the first - the papers contained by the apparatus are its focus - so it probably doesn't move us much further in our attempt to understand what we mean by the word 'file', we have more or less the same problems about empty files and blank papers as before. So we come to the fourth item in the list and here we enter the electronic era. At least we have escaped the traps inherent in the physical material manifestation definitions. But have we? We have introduced a rather tricky problem with the word 'store' if we want to abstract our concept from physical reality. That is, if we regard 'related records' in purely abstract terms, as 'items' of information, in what sense are they 'stored for use by a computer' - as zeros and ones, as electrical potentials? Trying to tie the concept to computer 'memory', another delightful fudge, introduces the problem of delimiting the concept. If 'memory' is made up of zeros and ones then how to we delimit our notion of 'a file' - surely a computer's 'memory' is just one long string of these digits? Does that make the computer's 'memory' just one large file? If not then it is clear that some zeros and ones are 'different' from others of the same ilk. Reducing our thinking to the individual electronic components that make up the 'memory', a seemingly necessary implication of the word 'store', clarifies nothing, and in fact just adds another layer of confusion.

So if our attempt to ground our concept in space and time has failed, then our attempt at abstracting it from 'material reality' hasn't got us out of the mire either. Defining a file as any collection of related 'items' of information introduces the problem of 'relatedness' - in what way are they related. Staying in 'information' mode restricts the type of relationship to 'conceptual'. Yet I have masses of information in my head, that are all conceptually related at all sorts of levels, so is my mind one large file? Not only does using the concept of 'file' in the context of mind seem strange, to make the least of it, but the idea of conceptual relationships acting as file defining elements reintroduces the problem of delimiting a file.

Despite all this definitional thrashing around we all know *exactly* what the word 'file' means. This must imply that 'meaning' is a much more powerful aspect of living than can be captured by the formal means that linguistic method, definition, provides. One can't help thinking that any attempt at definition is blatant reductionism. You have taken 'meaning' out of its living context and thereby killed it. Meaning can only exist in terms of the whole shebang, I can only acquire it in those terms through experience, no formal definitional process can give it to me because the associations between concepts are infinite. I can't have a sensible meaning for the concept 'file' without one for 'information', for 'related', for 'item', for 'store', for 'memory', and so on. Moreover each one of these concepts carries its own infinity of associations. By rights we should all be bogged down in a mass of definitions, but the fact is that we aren't! Thinking is a product of living, not processing information, meaning is processing experience, no definition can possibly work in the absence of any experience. The language of life is based on pattern not definition because meaning derives ultimately only from experience.

There is an obscuring force in the modern idea that the human condition is a result of information processing because we don't yet have a reasonable understanding of what information actually is despite its critical role in our modern way of life. We are like the Iron Age blacksmith who is the acknowledged expert on the production of iron artefacts, but who cannot answer the question "what is iron?"

> Indeed, he has no frame of reference within which to even begin to understand what it is you are asking! To give the kind of answer that would satisfy you, he would need to know all about the molecular structure of matter - for that surely is the only way to give a precise definition of iron. (Or maybe there are other ways, ways that require theories we ourselves are not aware of - This possibility merely strengthens the point I am trying to make.) But not only is your man not familiar with molecular theory, he probably does not even conceive of the possibility of such a theory!
>
> (Devlin 1995, p. 1)

The expertise of the modern programmer like the expertise of the iron age blacksmith is based on experience, not a theory, or even a simple definition of, information, because we don't yet have one! Like the example of the word 'file' discussed above, we all *know* what information is without being able to explain the concept. This is the power of pattern language - meaning out of experience.

Our abstractions, our webs of meaning, are fundamental to thinking, but not themselves, the source of thought. A computer program executes on a machine but it is produced by thinking, programming *is*, in essence, *thinking.* The context for programming in therefore life, *not* the machine environment - the program can't execute without that environment, but, equally, it can't be *produced* in that environment. Meaning is required, not just symbol processing - the code is all symbol, it exists in the machine environment. Using the code symbols for understanding *is* possible but it takes a great amount of experience, and, in any case, the programmer is processing the meanings, not executing the code, and is therefore using the symbols differently compared to the way that the machine does. It is precisely this difference that causes the phenomena that we know as 'bugs', humans think, that is process meanings, machines don't, and are therefore incapable of such misunderstandings, these errors of logic. The program is merely the means by which human thinking about the problem being dealt with is converted into a form that can be processed by the machine.

It is this fact, that programming is a form of thinking that gives rise to the idea of bringing the language that the computer uses closer to 'natural' language. A great deal of effort in computer science has been put into the task of bringing programming languages closer to the style of natural language on the basis that this will reduce the difficulty of programming. But the simple fact is that attempts to empirically confirm this hypothesis have failed.

> Given the small sizes of and inconsistencies among the reported effects, it is not even clear that notation is a major factor in the difficulty

> of programming. The study of programming languages has been central to computer science for so long that it comes as a shock to realize how little empirical evidence there is for their importance. Second, many of these effects tend to disappear with practice or experience. This raises some doubt as to whether these results reflect stable differences between notations or merely learning effects and other transients that would not be significant factors in actual programming performance.
>
> (Sheil 1981, p. 108)

There are two aspects of this conclusion that are of interest here. Firstly, the widespread intuition that more 'natural' languages should be easier to program with has to be based on something, and that 'something' is probably the proponent's own perception of the difficulty that she has with using the less 'natural' notations, or, more likely, her perception of the difficulty that she sees novices having. This relates to the second point of Sheil's summary, that notation does *seem* to have some beneficial effects in the novice situation. Tests with two simplified languages in 1973 found the IF statement to be easier to use than FORTRAN arithmetic *for novices* (Shneiderman 1976*b*).

The introduction of structured programming techniques to address the reading comprehensibility issues caused by the unconditional transfer of control from one place to another in a program using 'goto' statements resulted in several attempts to measure the expected benefits such as ease of understanding and modification of code. For example, Lucas and Kaplan, ran an experiment using students of a Stanford Graduate School of Business course entitled Information Systems Technology and found that their results "provide some support for the advantages claimed for structured programming, especially for the ease of modifying structured programs and the efficiency of the resulting code" (Lucas & Kaplan 1976, p. 138). This conclusion supported similar results obtained by (Sime et al. 1973), (M. E. Sime 1977) and, (Green 1997) using similar methodologies. Another study, based on understanding programs written to differing levels of structure found that the main benefit of the more structured code was in the increased confidence that the programmers had about their understanding rather than in any more 'objective' measure of comprehension (Weissman cited in (Sheil) 1981). Similarly, Sheppard et al noticed that to significantly degrade performance in memorising, modifying and debugging programs, the 'structure' had to be verging on the deliberately chaotic (Sheppard et al 1979).

None of this suggests clear cut support for Shneidermann's view that "the choice of control structure does make a significant difference in programmer performance" (Shneiderman 1980, p. 81), or that structured programming techniques must aid programming. As Curtis points out, "relevant empirical findings are mixed, due to the variability introduced by individual differences, languages, tasks, and so on (Curtis 1990). Moreover, Green's (Green 1997) and Sime's (Sime et al. 1973) results, in particular, suggest that the effect of practice on performance, that is, experience, clearly overides any effect caused by changing notation. This 'practice effect' even shows up in the comparison of static and dynamic typing, for although the dynamically typed language produced more errors, they were mostly related to representing data rather than the dynamic nature of data types, and their occurrence tended to reduce with increasing practice (Gannon & Horning 1975). What the 'practice effect' suggests is that the effect of different forms of notation will show up, not in programmer performance, but in the development of individual programming skill. That is, most of the measured differences in performance relate to the way that a program is 'modelled' in the mind of a novice, as compared to that of a master.

> Undergraduates and other novices do not have the same structures (often called schemas) built up is long term memory as do experienced programmers. Thus, their reasoning about programming is qualitatively different from that of more experienced programmers.
> (Curtis 1982, pp. 213-4)

What we really need to know is the effect that notation has in assisting the structuring of knowledge that underpins the learning of the skill. The failure to demonstrate an effect on programmer performance is, in these terms, irrelevant, except insofar as it points to the difficulty of measuring cognitive effects generally. If it is not possible to measure effects on programmer performance, it is hardly likely that it is any more feasible to measure effects on learning.

What we are suggesting here is that what is needed is not a different notation in the same vein but a different notation in kind. A programming language remains, in essence, a programming language, a system of logic, no matter how close you manage to bring it to 'natural' form. It is not the linguistic form as such that causes the difficulties of programming, and this is backed up by the results discussed above, but its logical nature. Fiddling with the notational form doesn't change its basic nature.

8.4 The Measurement Problem

A major problem with the idea of basing pedagogical practice on empirical data is knowing exactly what it is that is being measured by the data. It is not yet clear that we even know how to assess quality in program terms, let alone programming or programmer terms. When, as Yourdon points out, the question, 'what are the qualities of a good program,' elicits a plethora of different answers, the idea that we can assess programmer performance in any meaningful way is clearly ridiculous (Yourdon 1975, p. 6). Yet how else can one assess the efficacy of a teaching method other than by some programmer performance metric. Assessment in introductory programming courses tends to default to more or less simple measures of programming knowledge, rather than performance, not only because of the difficulty of measuring performance, but because of the difficulty of controlling the opportunities for cheating as well. But even if the environment in programming tests could be regulated, assessment of skill is still an issue. Experiments conducted on experienced professional programmers showed that the measures of performance covered a range from 5:1 to 28:1 in magnitude (see Table 8.1). It can only be presumed that the employers of these programmers were satisfied with their performance in the field, so the differences demonstrated in the experiment clearly indicate the difficulty of devising any objective measuring technique.

Performance Measure	Worst	Best	Ratio
Debugging Hours - algebra program	170	6	28:1
Debugging Hours - maze program	26	1	26:1
CPU sec. For program development - algebra program	3075	370	8:1
CPU sec. For program development - maze program	541	50	11:1
Coding hours - algebra program	111	7	16:1
Coding hours - maze program	50	2	25:1
Program size - algebra program	6137	1060	6:1
Program size - maze program	3287	650	5:1
Run time (CPU sec.) - algebra program	7.9	1.6	5:1
Run time (CPU sec.) - algebra program	8.0	0.6	13:1

Table 8.1. Comparison of Programmer Performance

The reaction to these results has been rather simplistic. Even someone as experienced and perceptive as Edward Yourdon is led to say that it shows that some programmers are very good and some are extremely bad. But I don't think that you can even draw as straightforward a conclusion as that from this experiment. There are simply too many factors that feed into performance that have not, and probably cannot, be controlled for. Presumably all these people came from different programming environments and it is not clear from the information in the paper that the experimental test involved using a programming system that they were all equally familiar with, or even that the best and worst performers in one measure were the same individuals in the other measures.

But even if familiarity with the test programming system was equal across the participants, it would still be the case that they would all be working in different kinds of conceptual contexts, so that, for example, a programmer who was working on some sort of graphical software would probably find the transition to the maze problem easier than someone whose normal working environment was accounting software. So the experiment was really a measure of the fit between the individual 'psychological field' of each participant, consisting of, among other things, background, experience and so on, and the particular tests and the environment in which they were conducted. In pattern terms we would say that it was a test of the fit between the personal pattern language of the programmer and that required for optimal performance in the test.

Take a simplified example, the testing of rat intelligence by seeing how long it takes a rat to learn to run a maze. As the devious organiser, I have presented my experimenter with two rats, one which I have freshly caught in the city, the city rat, and one which I have caught in the country, the bush rat, without telling her anything of their different backgrounds. My experimenter finds, not surprisingly, that one rat learns the task very much faster than the other and concludes that the range of rat intelligence is huge. Of course the test shows nothing of the sort. One would expect the city rat to learn a maze more quickly than the bush rat because the maze environment is closer to the conditions that it is used to. The bush environment does not generally consist of impenetrable continuous barriers, any obstructions in such an environment are likely to be discontinuous and to be of a nature that allows one to push one's body through them. In other words the experiment was really measuring the fit between the experiential background of the rats and the test environment, not their intelligence in any abstract sense[4]. So in the case of the programming test, just to mention one possible scenario, it might have been the case that some of the spread was due to the simple fact that the best performers in each test had written a program for a similar problem previously. There is no way to control for such confounding factors, even asking the programmers to reveal if they had encountered similar problems is not entirely certain, and therefore the results have to be treated cautiously, particularly insofar as they are read as a measure of individual programming performance in the general sense.

It is important to remember always that experts are human, that their expertise is merely a part not the whole of their being. Testing their performance in the area of their expertise implies that it can be treated in isolation, that they are, temporarily, 'programming mechanisms' not human beings. Dr Strangelove types excepted, it is not normal to expect generals to be *total* warriors, "fighting machines" pure and simple. They will have at least some of the same ambivalence to the realities of war as everyone else, so even their military judgement will be affected by their personal view of the world. Personality is largely established by the start of a career, so it is difficult to believe that the dynamics of their field can dominate over their personal opinions and feelings.

> Every expert is a human being; and technical opinions reflect the political views of those who give them. Generals and admirals are confident of winning a war when they want to fight; they always find decisive arguments against a war which they regard as politically undesirable.
>
> (Taylor 1964, pp. 124–5)

The technical aspect is just one aspect of a person's experience, albeit an important one, and the fact that there is a dynamic relationship between expertise

[4]Of course, any talk about intelligence begs the questions as to what it actually is and whether or not *any* test for it can escape from cultural or intellectual bias of one sort or another.

and personality works both ways. For example, this idea that a person brings their particular experiential background with them into any unfamiliar situation, can be seen as existing as a wider concern in social affairs, in terms of 'conflict of interest' issues, and the fact that it is widely believed that legal decisions are better made by juries, that is, a number of people from a diversity of backgrounds, rather than by judges, whose view of the world is likely to be somewhat conditioned by their constant involvement in the legal system. So the idea that programmer performance can be treated as a factor to be studied in isolation from the whole personality is somewhat dangerous. This was not entirely overlooked by the experimenters themselves, it must be said, but it has been a feature of the response to their results. One of the experimenters suggested elsewhere that the reason "experienced programmers vary by an order of magnitude for most performance variables"(Sackman 1970, p. 47), is that the programmer's wider personality is an issue in performance measures and this needs to be taken into account.

Several attempts have been made to investigate the effect of particular traits on human-computer interaction - assertiveness and the perception of the amount of control that a person has over their environment as factors in preferring batch over time-sharing systems (Lee & Shneiderman 1978), aggressive - humble and introversion - extroversion measures against results in an introductory programming course (Newsted 1975), analytic versus heuristic cognitive style in relation to the differing decision structures of various information management environments (Zmud 1979), need-achievement and evaluative defensiveness as factors in predicting performance in decision systems (Wynne cited in (Lee) 1978 p. 563), and so on. While it is undoubtedly true that personality is a factor in problem solving and decision making situations, attempts to associate particular traits with performance in these tasks will probably always lead to unreliable and inconsistent results due to the sheer complexity of the mix of factors.

> Despite the potential importance of cognitive styles for management decision-making and information use, conceptual and methodological weaknesses have sharply limited the payoffs from this line of research
> (Taylor & Benbasat 1980)

But it is not just the "conceptual and methodological weaknesses" of the research that is a problem here, it is the underlying metaphysical assumption of Descartes' separation of *res extensa* and *res cogitans.* As beings we are obviously whole systems, yet the behaviourist refusal to accept anything but overt behaviour as a measure of psychological state (Skinner 1953, p. 35) still colours social science research as if "the process of feeling", in Susanne Langer's sense, was not both objective and subjective in its effect.

> "Behavior," "stimulus," and "response" are working notions of the animal laboratory, generalized and stretched in the hope of covering the whole field of psychological facts; but beyond the context in which

> they were originated - experimentation on animals - they quickly decline in usefulness. A term that *designates* a vast variety of phenomena cannot be used to describe their differences, let alone to account for them. Abstractions do not designate phenomena at all, but serve to describe them. There is no object or event called "gravity," but such phenomena as the flow of water downhill, the position of stars in relation to each other, the attraction of a compass needle to the magnetic pole, are widely diverse events or conditions describable by use of the concept of gravity. There are such things as stimuli and responses; to isolate and label them, even to pair some very simple ones, is a sort of taxonomy; it does not furnish any principles of analysis or construction, any terms to describe the relations between observed events. [Emphasis in original]
>
> (Langer 1962, p. 5)

The transfer of these elements of behaviourism from the study of animal behaviour has mired Social Science in mechanistic thinking, when, indeed even Skinner himself "has to smuggle in mentalistic assumptions in order to make sense of his simplest animal experiments" (Flanagan 1991). There is one indivisible process going on in any human behaviour, it is not tenable to make a statement like "nerve impulse is converted into thought" without implying that mental function is some sort of "mysterious transubstantiation" from one type of substance, *res extensa*, to another, *res cogitans*. Nerve impulse is "*felt* as thought", both are aspects of the same 'whole' system, that's the point (Langer 1962, p. 10).

> The expression "felt as thought," which is here substituted for "converted into thought," raises another issue, the power of a new concept to concatenate the findings in a general field of research. "Feeling" in the broad sense here employed seems to be the generic basis of all mental experience - sensation, emotion, imagination, recollection, and reasoning, to mention only the main categories. Felt experience is elaborated in the course of high organic development, intellectualized as brain functions are corticalized, and socialized with the evolution of speech and the growth of its communicative functions. On the other hand, the mechanisms of felt activity are heightened forms of unfelt vital rhythms, responses, and interactions; a psychology oriented by this concept of feeling runs smoothly downward into physiology without the danger of being reduced to physiology and therewith losing its own identity. Even if it should ultimately appear as a branch of physiology, the area of its branching is likely to remain quite visible, though without a sharp dividing line (there are very few such sharp lines in nature): it is the area where vital (probably neural) processes begin to have psychical phases, i.e., to be felt. We may not always be able to judge what activities are felt; such judgments, with respect to speechless creatures, rest on many speculative grounds, not

> only analogies between animal and human behavior, but especially phylogenetic continuities and structural homologies.
>
> (Langer 1962, pp. 10–11)

It is reality that keeps us grounded, that does not allow the mind to totally disassociate, that keeps us 'whole' in fact, just as it was reality that made us what we are. Abstraction is a tool for life, not a mode of life, "our constant sensory stimulation, even without conveying any new information, serves to keep us realistic in waking life to the extent of not letting the brain freely hallucinate as in dream"(Langer 1962, p. 19). But this wholeness invalidates any sort of reductionist research and we encountered this limitation in our own attempts to measure the effectiveness of patterns in helping novices learn to program (See Section 10.5). The fact of the matter is that any attempt to measure programming performance devolves to an assumption, namely that the relationship of 'performance' to the whole 'psychological field' is both simple and static, when even the attempt to divide the task of programming into cognitive subtasks itself, turns out to be very problematic. If we can't even establish clearly, by experimental means, that program debugging requires comprehension (Bishop-Clark 1995), that code modification relies on a combination of comprehension, composition, and debugging (Koubek et al 1989), or that program comprehension involves debugging, modification and learning (Shneiderman 1976*c*), then relating programming performance to psychological factors in any sensible fashion is likely to be well beyond our means.

All this means that we are, at least for the present, stuck in the realm of theory rather than empirical modelling, but this should not deter us as "there are many models in science that clearly do not purport to have relations to an empirical system" (Downes 1992, p. 143). Theories, in this sense are "candidates for psychologically real representations and rules" (Gopnik & Meltzoff 1997, p. 33), rather 'empirically real'. One of the main drivers of theorising about the difficulties of programming has been the idea of 'cognitive load'. Cognitive load is a theory that derives from Miller's insight that working memory is limited to about seven 'chunks' of information, and that they can only actively process two or three elements at a time (Miller 1956). Most of the strategies that are commonly seen in problem solving situations in virtually all domains, problem clarification, breaking a problem down into key elements, drawing or modelling the problem domain, simplifying or redefining the problem, etc. can be seen as ways of addressing cognitive overload. More importantly, however, these restrictions on the processing power of the brain suggest that human reasoning is based on something more than just 'working memory', that to be as effective as we are there must be some other factor involved. As the example with Chess discussed in Section 6.2 (in particular, see Figure 6.3), expertise in a field is based on knowledge that encompasses much more than can be accommodated in a mere seven chunks, in fact it has been shown that Chess masters have as many as 100,000 board configurations at their disposal (Simon & Gilmartin 1973). Clearly the limitations of the immediate working memory space are being avoided by the use of long term memory.

But this introduces the problem of how the expert has such immediate access. Accessing stored information involves being able to find what you need in the mass and this implies that it is a well organised store of information. But there is an even more profound implication involving how it got to be so well organised. The only way that such a store of information could get to be both large and well organised is for it to have incorporated the organisation as it was being built. Here the idea of groups of 'chunks' or 'schemata' enters the scene as a means of enabling the transition from working memory to long term storage and vice versa - "schemata have the function of storing knowledge and reducing the burden on working memory" (Garner 2001). Here, once more, what we have is constructivism, the development of the 'mind' of an expert, rather than any sort of 'random' storage of information, knowledge as a 'web' of 'ideas' or concepts.

> Knowledge ... isn't a copy of reality ... it's a reconstitution of reality by the concepts of the subject, who, progressively and with all kinds of experimental probes, approaches the object without ever attaining it in itself.
>
> (Piaget in (Bringuier) 1980, p. 110)

Transcending the limitations of the animal brain, or even the pure neuronal dynamics of the human version, involves another one of those epistemic breaks, it is simply impossible to explain the mind in terms of the basic elements of brain activity. So although animals display some degree of abstractive and metaphoric ability - an animal does not require a 'perfect' fit between its stored representation of a predator and some presence in its current experience, for example - this is far coarser than that displayed by humans. So a seagull chick responds to the presence of an adult with a begging response, but it will equally respond to a beak that has been detached from a dead seagull, and even to a stick that only vaguely resembles a beak (Tinbergen 1951). This demonstrates that it has abstracted enough of the features of 'adult beak' to trigger the response, but that it can be easily fooled. The behaviour is still tied closely to brain dynamics, that is, it is largely instinctual, and therefore the animal is not capable of those discontinuous 'leaps' that characterise 'mind', where response has been liberated from the brain's own continuous neuronal dynamics. Language is the ultimate expression of this liberation, of course, associations are now made using a means of accessing memory through pure symbol that is far more powerful than the percept-level association used by the seagull chick.

> Symbols, metaphor, and analogy recapitulate the ontogeny of a dynamic state through contestation with another, resulting in a new synthesis within the epistemological domains of those states that are consequently re-formed. Metaphor, symbol, and analogy all share the key characteristic of synthesis of apparent and, in more advanced utility, non-apparent elements of objects and events. A metaphor is thus

> a heightened degree of routine association made remarkable because it involves the juxtaposition of apparently dissimilar phenomena. This contributes significantly to our survival, enabling us to transfer solutions across problems with similar goal structures. In the most advanced stages of brain development, we achieve true system mapping, in which we far surpass other species in our capability to transcend perceptual similarity. Discerning correspondence in non-similar phenomena is one of our highest achievements. ...
> Constraints are integral to this selected cognitive strategy. A language is by nature open ended and capable of infinite combinations, but its semantic value would be null if there were no categorical restrictions. The variety and richness of semantics depends on the tension between the language as a means of leaping dynamic states while constrained by the inherited neurological processes upon which those states reside. Meaning, in large part, emerges from this tension, from the continual intersection of an abstracting system of organization that must relate by nature of its associative propensities to the external world.
> The metaphoric/symbolic quality of language production is inevitable because meaning must be constructed from the associations of often disparate elements and events. To say language or its component words are 'representational' misses a critical point: words cannot represent singular objects or events without recourse to a variety of associations. Essentialism is impossible in linguistic constructs. On the other hand, cognitive categories must relate to the external world, or an organism would not efficiently categorize (and thus survive in) its environment, in other words, the construction of categories must be evolutionarily, and consensually, viable.
>
> (Henry & Rocha 1996)

The cognitive abilities that underlie language thereby increase the potential of the neuronal substrate through their associative (pattern) and generative (language) power. And if this is the way that the limitations of neural dynamics were transcended, then it suggests a clue to a way of dealing with the limits of the cognitive system. Presumably there is some sort of relationship between cognitive load and brain dynamics. At some level the demands on the brain of the creation and use of abstractions overwhelm even the increased potential that they supply - that is, the cost outweighs the benefit.

> Deciding to create and use an abstraction is not costless. There is a cognitive overhead merely in exploring the issue. There is also an overhead, identified by the 'cognitive dimensions' analysis, in creating many abstractions, since maintaining them in future may prove quite effortful if they have to be redefined.
>
> (Blackwell et al 2002)

It is this cost factor that drives the creation of ever higher level abstractions, and explains the difference between novice and expert performance. What novices are doing is *creating* the web of associations, the pattern language, because that is what learning is. So most of the cognitive load for novices in their programming tasks is the cost of this setting up process. And this is also where the drive for ever increasing degrees of abstraction comes from. If abstraction works to help overcome the limits of the neural system then it should also work at the task of assisting overcome the limits of the cognitive system, because the cognitive system is just the neural system at one step of abstraction. The next step is a sort of abstraction of lower level abstractions.

This can be seen in pattern language evolution, where a structure that is a pattern language at one stage of a person's development becomes a single pattern in a developing larger language. We discuss this further in Section 5.9 in terms of a child's acquisition of natural language as levels of 'linguistic organisation', and explicitly as pattern language evolution in Section 7.6. The likely memory resources involved are covered in Section 9.4. It is this process, by the way, that lies behind the 'jargon' phenomenon, experts are using concepts that have been assimilated into their own thinking but which are meaningless to someone who doesn't possess them. The way that the expert explains the jargon to the initiated is to go back to a lower level of abstraction, that is, to where the concepts being used are likely to be more familiar - one is breaking the single pattern back down into the form it had when it was a pattern language in one's own mind.

Cognitive load, then, is an expression of homeostasis, where homeostasis is "the capacity of a system to hold its critical variables within physiological limits in the face of unexpected disturbance or perturbation" (Beer quoted in (Hall) 2005). The critical variable being controlled in this case is the system's own organisation, its ability to maintain itself as a coherent system of knowledge. "A cognitive system is a system whose organization defines a domain of interactions in which it can act with relevance to the maintenance of itself, and the process of cognition is the actual (inductive) acting or behaving in this domain" (Maturana 1970, p. 13). The cognitive system is about 'conceptual organisation'. If it becomes overloaded the result is a breakdown of the organisation that is the *very* reason for the system being in existence. In other words its sense of wholeness, its internal coherence, is gone because organising one's perception of the world is its sole purpose in terms of survival.

There are no survival implications in most situations where cognitive load is likely to occur, of course, nevertheless the loss of cognitive coherence has to be a negative factor in terms of any intellectual performance. Dealing with complicated situations involves both strategic and tactical level thinking (Lewis 1981, p. 87), but the later is much more effective if it is based on an overall strategic view. So the main effect of cognitive overload is probably to effect the strategic level view, leading to uncoordinated and misdirected activity at the lower level, which is exactly the type of behaviour that we see in unskilled performance. But this effect is triply pernicious because the resulting erratic activity itself adds to the cognitive load, and moreover, must tend to inhibit the development of a better strategic view.

Chapter 9

The Psychology of Learning Programming

Memory and imagination are but two words for the same thing.

Thomas Hobbes (17th century philosopher)

We have - despite what psychologists, pædagogues and the like may think - not the faintest idea how knowledge, insights and habits are transferred. It is not unlikely, that the actual transfer is always by imitation, and that all the explicit teaching in the scientific tradition is no more than giving the student some verbal handles, which are no more than an aid to memory. If this is true, then all purely "scientific teaching" - i.e. the explicit rules and no more - is bound to be, and to remain forever, a barren activity.

(Edsger Dijkstra 1982, p. 109)

9.1 A New Way of Thinking

Teaching anything is one of the great challenges inherent in the human condition, because nothing is ever really *taught*, it is *learned*. Someone either learns what is put in front of them by the teacher or they don't, the actual performance of the learning is entirely in the learner's purvey, not the teacher's. In the end, the role of *teacher* is really that of *facilitator of learning* and nowhere is this clearer than in Plato's dialogues, where most of what Socrates does is to *question* his pupils. He causes them to think of the matter under discussion in terms of the beliefs that they already hold, and by this means moves their understanding forward through the apprehension of the dialectic of the contradictions exposed thereby.

> Ausubel (1968) states that "the most important single factor influencing learning is what the learner already knows" (p. vi). We must therefore search for an explanation of variation in human learning

> capacities primarily in the cognitive and affective experiences the individual has had. In the absence of organic brain damage and such occasional biochemical hereditary defects as phenylketonuria, most differences in human learning capabilities at any point in one's life up to old age should be predominantly the product of prior learning experiences.
>
> (Novak 1977, p. 57)

Of course, if this is true in explaining differences between individuals, then it must also drive any explanation of the individual's own level of learning capability at any particular time. One can't explain a difference in levels between individuals using a factor that does not relate to the capacity that is being compared. This boils down to the statement that an individual's capacity to learn a particular knowledge domain or skill is a measure of the relationship between the individual's previous experience in terms of the nature of the new knowledge or skill. Difficulties in learning a new field thus represent a 'bad fit' between the cognitive or mental structure of the learner and the conceptual structure of the new material, or, at the very least in the way that it is being presented. To a significant extent the 'cognitive structure' of an individual is a direct result of the 'conceptual structure' of the totality of her previous experience. This is why the notion of wholeness is so important in relation to learning, the present is a product of the past, and it is difficult to see how one can unravel them to any extent and still address the individual at any level of understanding.

Given this degree of connection between the past experience of the novice and their ability to learn anything, the idea of teaching programming begins to seem a little ridiculous. If programming is considered to be horrendously difficult, then teaching it would seem to verge on the impossible. As Gruenberger has said, "I have to conclude that I don't really know how to teach computing, and I'm reasonably sure that no one else does either" (Gruenberger 1977, p. 124). Yet teaching is just a process of directing, or assisting, the learning of another, and we all spend our lives learning. This is a constructivist approach, the idea that knowledge must be constructed by the learner, it cannot be supplied by the teacher (Bringuier 1980).

> The fundamental fact about learning ... [is that] anything is easy if you can assimilate it into your collection of models. If you can't, anything can be painfully difficult. Here too I was developing a way of thinking that would be resonant with Piaget's. *The understanding of learning must be genetic.* [emphasis in original] It must refer to the genesis of knowledge. What an individual can learn, and how he learns it, depends on what models he has available. This raises, recursively, the question of how he learned these models. Thus the "laws of learning" must be about how intellectual structures grow out of one another and about how, in the process, they acquire both logical and emotional form. ... [What is required is] an applied genetic epis-

> temology expanded beyond Piaget's cognitive emphasis to include a concern with the affective. It develops a new perspective for education research focussed on creating the conditions under which intellectual models will take root.
>
> (Papert 1980, pp. vii–viii)

So it is the disjoint between the models in the learner's mind and the models underlying programming that cause the difficulties that learners exhibit. The novice's past has not equipped her to deal easily with the strict symbolic logic involved, so she is basically learning from scratch, learning a new way of thinking, a way of thinking, in fact, that is not only different from her previous mode, but strictly contradictory. Those models and modes of thinking that have been laboriously constructed out of experience are rendered useless by the current models used in the teaching of programming because they are based on *meaning*, and, as we saw in the case of Helen Keller (see Chapter 5), symbols without meaning are empty, mere 'signs', hardly worthy of being "called a thought" (Langer 1976, pp. 62-3). Moreover if this is true then the solution to the problem is quite clear - the current mode of teaching programming *must* be changed, as changing the mode of thinking brought to the task by the beginner is simply not an option - short of rearranging the common experience of life for everyone, anyhow.

The hidden 'faith' of education is that every human, with the usual obvious caveats, is in a state conducive to learning virtually anything. As the California State Board of Education states in respect of their mathematics education:

> These standards are based on the premise that all students are capable of learning rigorous mathematics and learning it well, and all are capable of learning more than is currently expected. Proficiency in mathematics is not an innate characteristic; it is achieved through persistence, effort and practice in the part of students and rigorous and effective instruction on the part of teachers. ... The standards emphasise computational and procedural skills, conceptual understanding, and problem-solving.
>
> (CSBE 1999, quoted in (Macnab) 2000)

But this educative 'ideology' is a direct consequence of the way that we currently view the mind. It is based on human *understanding*, a property reliant on *meaning*. The fundamental principle of cognitive science that "the essence of knowledge is structure" (Anderson 1981, p. 5) is thereby constructivist in terms of learning - the only way to acquire knowledge is to build structure (construct). What this means is that we should be able to identify those elements of cognitive structure that contribute to 'thinking like a programmer'. The best way to approach this task is to examine the differences between the novice and expert minds, as what the expert has done is acquire, painfully, the cognitive structure that facilitates programming skill - the expert has learned the "new way of thinking" that the novice needs to learn. In other words the expert has constructed "meaning" out of experience, the only way that "meaning" can be acquired.

The learning of any skill involves a journey from novice to competency, and an appreciation of the psychological difference between the two end points is vital to any teacher. On the face of it, instructing a machine with a very limited repertoire of basic operations would intuitively seem to be easy, but history demonstrates that programming, and especially, learning the skill, is the cognitive equivalent of climbing Everest (McCracken et al. 2001). Understanding the cognitive structures on which competency is based, and how these representations of the requisite knowledge vary from those of the novice (Novik 1990), is therefore obviously desirable, however expertise is a complex and little understood state of mind. All that we can probably hope to do at this stage is to point to some differences in novice and competent performance, attempt to explain them in terms of various cognitive factors, and relate this to pedagogical practice.

Probably the most compelling difference between the two performance levels is in the apparent ease with which the expert proceeds compared to the floundering performance of the novice. “Experts seem invariably to know when to apply knowledge in a given task, whereas it is characteristic of novices that they often fail to apply what they know” (Chase & Ericsson 1981, p. 175). But it is difficult to account for this procedural difference purely in terms of the knowledge of the programming language involved as the novice can often answer questions that indicate a reasonably sophisticated level of such knowledge. What seems to be lacking is the ability to apply the knowledge, the factor that in the expert’s case appears almost automatic. But the the lack of this facility in the novice is one of the major blockages to learning because, if in dealing with a problem the student “cannot find his own way out he will not learn, not even if he can recite some correct answer with one hundred percent accuracy” (Dewey 1966, p. 160). Clearly the expert has some cognitive support structures on hand that act to change static knowledge into procedural form virtually automatically. This ability to convert static knowledge into practice almost without effort is known as ‘automatism’ and it is this feature that produces the observed differences in performance. The most likely explanation of the effect is that automatism reduces demand on limited resources, probably memory (Shiffrin & Dumais 1981, p. 139).

Moreover, we know what the “development of automatism” involves, namely, lots of practice. “Almost always, practice brings improvement, and more practice brings more improvement” (Newell & Rosenbloom 1981, p. 1). The obvious catch here, though, is that it is difficult to practice when the source of your difficulty is generating procedure from knowledge. Improving your knowledge base won’t help if you already have the knowledge. What is needed is a mechanism that acts to convert knowledge into action as the ‘automatism’ of the expert does. But providing such a mechanism is presumably what we think we are doing when we work through examples in our teaching materials and presentation. Clearly it is not enough to facilitate the practice that novices need, probably because the active component, the actual workthrough of the example, is not available to the student at the time it is needed and the written version is a pale imitation of the dynamics of an actual demonstration. So what we need to do is to find a way to make the written version more procedural in flavour, to give the novice the equivalent of the semantic structure that the expert has developed through experience.

In this chapter we want to explore the relationship between the expression of a skill and memory and relate it to the use of a pattern language to facilitate the practice that novices need but find difficult to do. The pattern language idea is relevant in this context because it arises in a situation where the amount of detail to be assimilated into the design of modern living space overwhelms the capacity of even experienced designers in the same way that program design problems overwhelm novices.

> The very frequent failure of individual designers to produce well organized forms suggests strongly that there are limits to the individual designer's capacity.
> We know that there are similar limits to an individual's capacity for mental arithmetic. To solve a sticky arithmetical problem, we need a way of setting out the problem which makes it perspicuous. Ordinary arithmetic convention gives us such a way. Two minutes with a pencil on the back of an envelope lets us solve problems which we could not do in our heads if we tried for a hundred years. But at present we have no corresponding way of simplifying design problems for ourselves. These notes describe a way of representing design problems which does make them easier to solve. It is a way of reducing the gap between the designer's small capacity and the great size of his task.
> (Alexander 1964, pp. 5–6)

This idea that a way of representing the relationships between concepts (this is what a pattern language diagram is) can be a useful tool in supporting reasoning tasks is supported by recent findings in differential psychology research (Monaghan 1995, p. 33).

The obvious strategy in considering the efficacy of patterns in introductory programming is to test the use of patterns under experimental conditions. Unfortunately, this turns out to be rather more difficult than might be expected due to the difficulty in controlling the multifarious variables involved. In our first two attempts to measure improvement in a group exposed to a pattern language and patterns against a control group we found that neither methodology tried provided clear cut evidence. In both cases the pattern group did perform better but we feel that we were unable to isolate the results from factors like the Hawthorne effect (Draper 2005), the 'practice effect' (Sheil 1981, p. 106) and other possible sources of improved performance. In the first instance we ran the test as an experiment completely isolated from the normal coursework (see (Porter & Calder 2004)) but found that motivating the required extra-curricula involvement difficult. Accordingly, the second attempt involved the use of extra help sessions for students undertaking a second programming topic, however maintaining consistent attendance again proved a problem. A third attempt, based on running two of the normal practical streams along pattern lines, was made, but providing a meaningful pattern context proved impossible in the restricted circumstances of the practical laboratory session.

So although it is obviously desirable to establish the benefit of using pattern languages in helping novices to become competent programmers it is also useful to clarify why it is that we feel they should be of such benefit, and that's what we attempt in this chapter. There is nothing new in this, pedagogical change is rarely based on solid empirical evidence simply because the educational context changes too fast. But, in fact, this is just a reflection of the way that the world is. Changes, important changes - think of the introduction of television, or in more recent times, cellular phones, for example - in the way that society works are made, or even just happen, without any attempt at trying to assess the ramifications. Reflecting this, it can be seen that most of the developments in programming languages and methodologies have been explicitly predicated on the belief that they will improve the performance of programmers by addressing factors that are essentially psychological in nature (Sheil 1981, p. 101), rather than as a result of empirical research. Indeed where such research has been conducted, usually in retrospect, it has "found few clear effects of changes in either programming notation or practice" (Sheil 1981, p. 101).

Given the difficulties encountered in studying the novice programming environment, then, it is clear that research into simpler and more mature, and therefore hopefully better understood, problem solving environments should at least enable us to theorise about some of the cognitive structures and processes involved in learning to program. In itself, the pattern idea has had an impact on programming that can be regarded as pedagogical in nature. It is clear that the patterns examined in the GOF book were, in fact, in common use prior to its publication, they "are solutions that have developed and evolved over time" (Gamma et al. 1995, p. xi). As Linda Rising points out "experienced designers read these patterns and usually remark, 'Sure, I've done that - many times!"' (Rising 1996). So the main import of the GOF book was to make programmers conscious of patterns that they were probably already producing incidentally in their code, in order to facilitate and broaden that use - it was a way of sharing knowledge in a readily usable form. So it is the *pattern idea*, not the individual patterns as such that was being introduced and the evidence is that it did enable us to think more clearly about what we were doing.

It is in this spirit that the idea of using patterns in the learning context arose. If patterns were useful in enabling experienced programmers to better understand what they were doing then this effect, on its own, has to be advantageous in the novice situation. In this field we need as much of artistic intuitive feeling as we do of basic knowledge - this is an art. As with music, theory cannot drive the composition of the artefact in the same way that scientific theory does a technological project. In other words it can't be formalised or 'engineered'. So the problem seems basically to be that when we are programming we are "forcing our interactions into the narrow mold provided by a limited formalized domain" (Winograd & Flores 1987, p. 75) which stifles the source of our creativity, the power of abstraction and metaphor. The programming domain provides the *tactical means*, the operators, to solve problems. What it does not, and cannot, provide is the insight behind the task of combining them to solve a particular problem, the *strategy*. "That strategic knowledge can be considered separate from knowledge of operators is shown by the fact that a solver may know *how* to apply either of two operators in a given situation but still not know *which* to apply" [emphasis in original] (Lewis 1981, p. 87).

The thrust of patterns has always been this strategic level dynamic, the cognitive aspect of using the operators to 'create' programs. And this relates to an aspect of advanced programming that *has* been studied. This is the use of 'programming plans' by expert programmers. These 'strategies', however, now appear to be "strongly tied to the particular learning experience of the programmer" and "related to the expression of design-related skills" (Davies 1990, p. 461). If nothing else, this points to the importance of the learning experience in shaping programming practice, and, if pattern languages really are an expression of strategic level thinking then the idea of using patterns in the teaching environment is one way, at least, of providing the basis for the use of such plans. But the real point about strategic thinking is that solving problems in any domain requires the same sort of insight, and produces the same sort of demand on cognitive resources.

Therefore, to explore the use of cognitive resources in programming we examine a situation in mathematics that causes similar difficulties for novices, and analyse a case study in the recall of digit sequences that suggests that the main source of novice difficulties is in the development and use of long term or 'semantic memory'. What the expert appears to have that the novice doesn't, is a mechanism, basically a network of connections based on understanding (Bransford et al. 2000, p. 127), a semantic network, in long term-memory that frees the short-term memory from the task of making the necessary connections on the fly. What we are observing in the novice are the symptoms of cognitive overload, the utilisation of the limited capacity of short-term memory, Miller's famous "7 ∓ 2 item" limit (Miller 1956), to do what the expert's pre-existing 'semantic network' does in its 'automatic' fashion.

9.2 A Parallel Example in Mathematics

Generating a proof for a geometric or algebraic proposition is a similar situation to designing a program to solve a problem. It involves a knowledge of the basic principles underlying the design space, the development of a clear understanding of the goal of the exercise and a plan for achieving it, and a successful implementation of the plan in the formal notation appropriate to the task. In both fields, mathematics and programming, it would appear the main difficulty arises in what might be called the planning stage involving coming to understand the problem and to plan its solution. Apart from things like mistaken notions of what constitutes a proof and invalid beliefs about deriving one (Recio & Godino 2001), the main difficulty that students of mathematics seem to encounter is at the level of understanding required to translate the proposition into a plan for proving it, that is 'designing' a solution.

> In two studies, Weber [2001, 2002b] observed eight undergraduates who had completed an abstract algebra course constructing non-trivial proofs about group homomorphisms and isomorphisms. The studies considered only those cases in which the undergraduates were aware of the facts and theorems needed to prove a statement and could construct a proof when specifically told which facts to use. Even in these cases, the undergraduates failed to construct a proof without prompting 68% of the time. Examination of these undergraduates' behaviors revealed that their strategies for constructing proofs were ineffective and crude. For instance, to prove a statement B, these undergraduates would often try to find any theorem of the form "A implies B" and try to prove A, even when the antecedent was implausible.
>
> (Weber n.d.)

So the difficulties that novices exhibit in mathematics show a striking resemblance to those exhibited by programming novices, suggesting that the cognitive models underlying both the skill itself, and the acquiring of it, are essentially the same. Understanding the concept space is the key so we need to distinguish between "learning that results in a relatively mechanical skill and learning that results in an understanding of the problem situation" (Anderson et al. 1981, p. 206) in general terms. The cognitive model indicated is one that provides the means for understanding problem situations in general terms because guiding the process requires more than static knowledge.

> Even if students are "logically capable" – that is, they know what constitutes a proof and they can reason deductively, recite and manipulate definitions, and draw valid inferences – this does not guarantee that they can construct anything beyond very trivial proofs. Knowing logical rules and the definition of a concept does not ensure that students can reason about that concept. Students often require an intuitive (conceptual) understanding of the concept that they are working with before they can construct proofs.
>
> (Weber n.d.)

Because the knowledge space involved in proof generation is, relatively speaking at least, well studied, it can serve as a reasonable basis for an attempt to analyse the cognitive models and processes involved, and this has been attempted as a qualitative study based on observing a 14-year old receiving instruction in geometry in advance of encountering it in the normal school curriculum, and by analysing interviews conducted with 7 students undergoing regular school instruction in geometry (Anderson et al. 1981). The interesting aspect of the findings of this research is in the resonances with the pattern language idea. For example, the researchers found "that planning exists as a logically and empirically separable stage of proof generation ... [and] that planning is the more significant aspect and the aspect that is more demanding of learning. Execution, while not necessarily trivial, is more mechanical" (Anderson et al. 1981, p. 192). This matches the thrust of Alexander's notion that design is like a hypothesis, "it cannot be obtained by deductive methods but only by abstraction and invention" (Alexander 1964, p. 92).

The picture of the process of transformation from novice to expert that emerges from this study is that of a journey from a declarative to a procedural form of knowledge. "In a declarative encoding, the knowledge required to perform a skill is represented as a set of facts. ... These facts are used by general interpretive procedures to guide behavior" (Neves & Anderson 1981, p. 60). Knowledge, for the novice, consists of the general rules of geometry and examples of worked-out proofs, both of which are presented in the instructional process. This knowledge is encoded declaratively for two complementary reasons - because that is how it is presented and because this form of representation is more concise. The point here is that the same set of facts "can give rise to a great many possible productions reflecting various ways that the information can be used" (Anderson et al. 1981, p. 214). Therefore using the declarative form of knowledge in generating a proof involves an intermediate step, translation from fact to action, what Neves and Anderson (1981) call "general interpretive procedures." This is the difficult part precisely because it is not 'mechanical' - there is no 'royal road' from fact to final proof, it requires the development of a hypothesis, the use of productions. " The data alone are not enough to define a hypothesis; the construction of hypotheses demands the further introduction of principles like simplicity (Occam's razor), non-arbitrariness, and clear organization. ... There is at present no prospect of introducing these principles mechanically" (Alexander 1964, p. 75).

But if the declarative form is both concise and ultimately flexible in that it is not tied to particular procedures, it has the major drawback that "its interpretative application is slow. Each fact must be separately retrieved from memory and interpreted. The interpretive procedures, to achieve their generality, are unable to take any shortcuts available in applying the knowledge in a particular situation. Many unnecessary or redundant tests and actions may be performed" (Neves & Anderson 1981, p. 60). So the novice's behaviour is based on a more or less unstructured search of the knowledge base. Typically a candidate postulate will be recited suggesting that the translation from fact to action is happening in short-term memory as fragments of the postulate are matched individually to the problem and developed into productions (Anderson et al. 1981, p. 215). The novice is making the connections 'locally' rather than retrieving the connected version of the knowledge from experience as the expert does.

The expert's 'procedural form' is dynamic rather than static, it is, in effect, the declarative version plus action, the conditions under which it is applicable are now built in (Whitehead 1929). It is a way of representing knowledge "as something that can be directly executed and so needs no costly interpretation phase" (Neves & Anderson 1981, p. 61). So learning, the acquisition of expertise, is this conversion of knowledge from a static to an active form. "Proceduralization is a process that eliminates retrieval of information from long-term memory by creating productions with the knowledge formerly retrieved from long-term memory built into them" (Anderson et al. 1981, p. 218). But this is an accumulative process, and it is the development of even larger compositions of 'active knowledge' that we detect as growing expertise.

> The effect of this proceduralization process is to enable larger composed productions to apply because the proceduralized productions are not limited by the need to retrieve long-term information into working memory. This in turn allows still larger compositions to be formed.
>
> (Anderson et al. 1981, p. 219)

However, it is important that the process from declarative to procedural form in memory is actually undertaken because there are some serious disadvantages to the production form. On its own, conciseness is an important virtue, but the really serious loss is flexibility because it is expressed in several ways. For example, the knowledge encoded in productions cannot be inspected, virtually the only way to understand the content of a production is by noting what it does, and this is the old problem of simulating logical progression, not something that we humans generally practice much in the course of our everyday lives. Moreover when a rule is expressed in a production it can only be used in that form, and the production cannot be changed (Neves & Anderson 1981, p. 62). In this sense a production is like a mould or template, it has little of the flexibility of a simple fact. But, because the production-based expertise has been developed over time from the original declarative form, the expert has access to both forms.

> Proceduralization and composition ... allow us to maintain the flexibility of representing knowledge in a semantic net and also to build production rules that will embody directly certain uses of the knowledge. The knowledge underlying procedures starts out as propositions in a network. Knowledge in this form can be changed and analyzed by the cognitive system. As one applies knowledge, the proceduralization process turns it into faster production rules automatically. Then composition forms larger units out of the individual proceduralized productions, in a gradual manner. These processes help explain some

> effects in the practice literature such as automatic speedup, development of parallel search, and inability to introspect on the application of well-learned procedures.
>
> (Neves & Anderson 1981, p. 82)

So what are the cognitive factors that underlie this "proceduralization" of knowledge that can help us facilitate it in students?

9.3 Memory and Meaning

It is an almost trivial observation that skill is highly dependent on memory in the sense that knowing how to do something is fundamentally remembering how to do it. This does not imply that skill is a completely memory-based operation, often the memory component of a particular task merely involves remembering what information is required and where to find it. But the fact is that it is meaningful patterns in long term memory that underpin knowledge-based expertise. This is shown by the studies of chess experts (Bransford et al. 2000) (Chase & Simon 1973) (de Groot 1966), and other knowledge-based experts (Chase & Chi 1980).

> We know that increasing experience and knowledge in a specific field (chess, for instance) has the effect that things (properties, etc.) which, at earlier stages, had to be abstracted, or even inferred are apt to be immediately perceived at later stages. To a rather large extent, abstraction is replaced by perception, but we do not know much about how this works, nor where the borderline lies. As an effect of this replacement, a so-called 'given' problem situation is not really given since it is seen differently by an expert than it is perceived by an inexperienced person.
>
> (de Groot 1965, p. 33–34)

The main difficulty in explaining expertise is accounting for the gap between the limited capacity of short-term memory as shown by Miller (Miller 1956) and the high-performance memory requirements of skilled performance (for example, chess grandmasters can recall hundreds of chess-board configurations - between 50,000 to 100,000 chunks according to (Simon & Gilmartin 1973)). Short-term memory span has been related to intelligence test scores (Bachelder & Denny 1977), and, more significantly shown to place severe constraints on problem solving and information processing performance (Miller 1956) (Newell & Rosenbloom 1981). As an example, correlations of between .80 and .90 have been established between aspects of working memory (digit span, mental arithmetic, etc.) and elements of reasoning (analogies, verbal reasoning, etc.) (Kyllonen & Christal 1990). It seems that transforming information, the main element of reasoning, places enormous demands on working memory (Holzman et al., 1982).

Yet the fact is that experts seem somehow to be able to bypass these limitations, suggesting that the major difference in performance in reasoning ability is due to differences in the amount of information maintained in working memory. What the expert seems to be doing is to use "their knowledge structures in semantic memory to store information during skilled performance of some task" (Chase & Ericsson 1981, p. 159) rather than relying totally on short-term capacity as the novice who does not possess them is forced to.

So 'semantic memory' is, if not the whole, an element of long term memory, and it is connected to knowledge in a general sense rather than to specific learning experiences (short-term or 'episodic' memory) (Tulving 1972). It is in structural terms that memory encodes meaning, the semantic network model, whereby concepts are linked in terms of their semantic relationships (Harley 2001, p. 283), and it is this structure that experts utilize in the performance of their skill. That this is so is dramatically illustrated in a study of a subject who became, over time, the apparent holder of the world record for the digit-span task (Chase & Ericsson 1981, p. 141), that is, the recall of strings of digits up to 80 digits in length. In 250 hours of laboratory practice over two years this individual increased his memory span from the statistical average of about 7 digits by a factor of 11, and the previous highest score recorded by a factor of four.

> An undergraduate (SF) with average memory abilities and average intelligence for a college student was paid on an hourly basis to participate in the experiment. SF was run on the memory span task for about an hour a day, 2 to 5 days a week, for over 2 years. The basic procedure was to read random digits to SF at the rate of one digit per sec, followed by ordered recall. If the sequence was reported correctly, the length of the next sequence was increased by one digit, otherwise the sequence length was decreased by one digit. Immediately after each trial, SF was asked to provide verbal reports of his thought processes during the trial. At the end of each session, SF was also asked to recall as much of the material from the session as he could. On some days, experimental procedures were substituted for the regular sessions.
>
> During the course of 25 months of practice, involving over 250 hours of laboratory testing, SF demonstrated a steady improvement in his average digit span from seven digits to over 80 digits. Furthermore, there was a parallel improvement in his ability to remember digits following the session. In the beginning, SF could recall virtually nothing after an hour's session; now SF can recall well over 90% of the digits presented to him.
>
> (Chase & Ericsson 1981, p. 143)

This example is important, not in terms of what it is - recalling large strings of digits is little else than a clever parlour trick - but for what it tells us about how memory works in relation to a particular skill. It is clear that the system developed by the subject is entirely based on setting up a hierarchical structure through relating the 'meaningless' strings of digits to an information system, a semantic network, that he already possesses and giving them meaning, and therefore making them useful in terms of the task at hand, recall, by that means. He is, in a sense, reusing a pattern language, or indeed several pattern languages, that he already has set up in his long term memory, for purposes entirely unrelated to their normal functionality. What it tells us is that skill is ultimately based on structural ordering of information, a web of meaning, in long term memory. The really surprising aspect of all this is that the semantic web can be used for a task that has nothing to do with its own raison d'être - it demonstrates the power of the relationship between meaning and process that meaning can be used 'secondhand' or 'out-of-context' in this way.

From the study it became clear that the subject was utilizing a set of protocols based on his prior knowledge of competitive running (he is a good long distance runner), ages, years, and other numeric patterns. He uses these as a mnemonic scheme for coding the digit sequences.

> By the end of 6 months - 100 sessions - SF had essentially completed his mnemonic scheme, and he was coding 95% of the digit sequences, of which the majority were running times (65%), a substantial minority were ages (25%), and the rest of the coded sequences were years or other numerical patterns (5%).
>
> (Chase & Ericsson 1981, p. 143)

The scheme works as follows. As the string is read out the subject attempts to relate the current subsequence to categories of running times, so, for example, 3492 would be coded as "three forty-nine point two, near world-record mile time". He has 11 major categories of running events, each containing a number of subcategories such as 'poor', 'very good', 'near record', 'average mile time for the marathon' and 'average work-out mile time'. Some sequences of digits do not fit any category in his running time scheme, and these are coded using ages, 'eighty-nine point six years old, very old man', or other common patterns such as years (1943 as 'near the end of World War II', for example).

The following diagram (Figure 9.1) illustrates the part of the semantic memory running-time net that is used in accessing the mnemonic encoding for 3492. His recall technique is very systematic, "he begins with the shortest race and systematically works his way up, category by category, with very few reversals. Within each category, he uses the same procedure of systematically recalling from the shortest to the longest subcategory, with pauses separating subcategories. At the lowest level within each subcategory, SF still generally recalls times in an orderly way from smallest to largest times" (Chase & Ericsson 1981, p. 150). Of course, this technique only covers the storing and recall of sequences in terms of long-term memory, it does not entirely explain how the limitations of short-term memory are overcome, that is, one can't assume that it is short-term memory that is holding the retrieval cues like 'near world record mile time' because this would account only for a maximum of 28 digits.

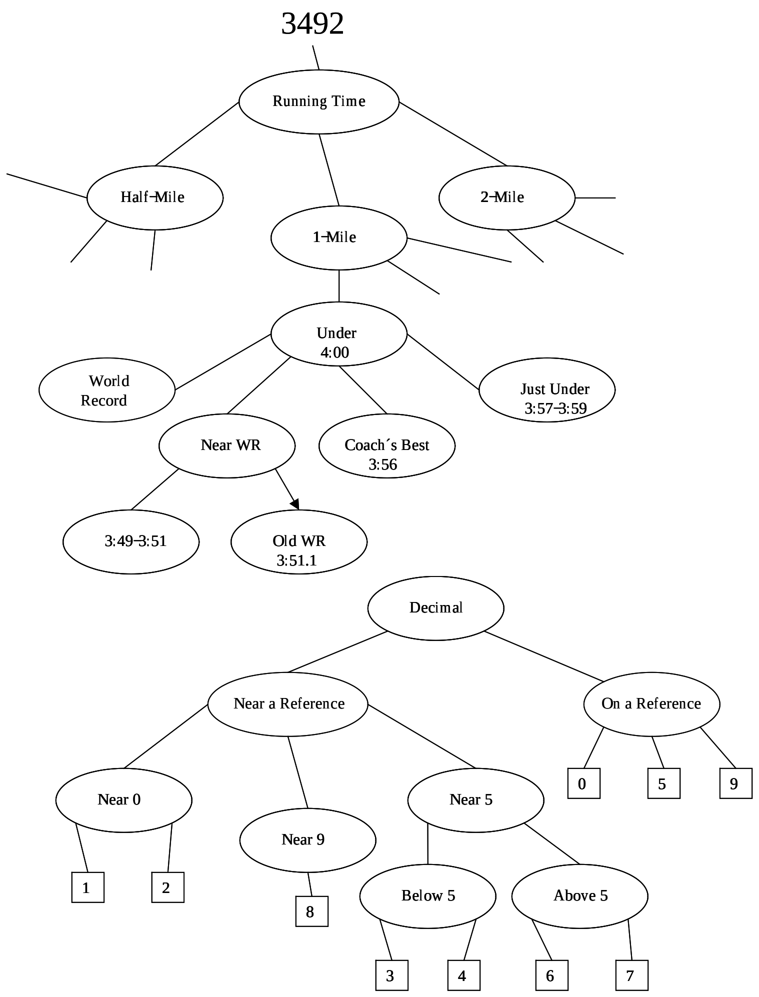


Figure 9.1: Section of the running-time semantic network for encoding 3492 (adapted from (Chase & Ericsson 1981, p. 164)).

There are two problems with this simple short-term memory retrieval model. First, the rehearsal-suppression experiments have proven to our satisfaction that SF's coded digit groups are not in short-term memory. Our analysis of SF's running short-term memory load indicates that only the most recent one or two groups occupy short-term memory momentarily while being coded into long-term memory.
The second problem with the simple short-term memory model of retrieval is that SF recalls too much. If we assume that SF's original memory span for symbols is around seven and he learns to recode single digits into groups of three or four digits, then his memory span should be around seven groups, or a maximum of 28 digits. In fact, because there is additional memory overhead associated with groups, the real memory-span limit is around three or four groups, or 16 digits. But SF's memory-span performance has increased steadily to over 80

> digits (22 groups), and there is no sign of a limit. There must be some other mechanism besides the mnemonic coding.
>
> (Chase & Ericsson 1981, p. 168)

This mechanism, it would appear, is a sort of "retrieval structure" operating between the short-term and long-term memories that allows the development of expertise. In all complex tasks, even seemingly trivial everyday activities like following a discourse, or watching TV, where visual and audial information has to be processed simultaneously and integrated in order to make sense of what is going on (Pezdek 1987, pp. 8–9), there is a need to store intermediate states. So, for example, in listening to a conversation, we have to be processing several levels simultaneously - handling what we are currently hearing, making sense of the previous part of the current sentence, fitting that to the meaning of the previous part, and keeping track of the task of comprehending the conversation as a whole. Until recently cognitive theory assumed that the intermediate stages in these various levels of the overall comprehension process were stored in short-term memory, but this would seem to exceed its capacity (Shiffrin 1976, p. 177).

At one stage it was believed that organised pieces, 'chunks', were stored instead of the individual elements of each, but as the group idea discussed above, and as Chase and Simon found with chess masters (Chase & Simon 1973), the capacity limit is still exceeded. It would, of course, be possible to postulate the existence of an intermediate level of memory on the computer analogy (immediate or sensory = keyboard cache, intermediate = cpu cache, and long-term = RAM storage) (Atkinson et al 2000) but this is a peculiarly 'physical location' type of classification that seems not to fit the functional characteristics of memory in skilled performance and tends to lead to a sort of infinite sequence of stage additions as apparent memory tasks increase in number and complexity. SF's retrieval structure, for example, has five levels (see Figure 9.2), are we supposed to think that he therefore has three levels intermediate between short and long term memory?

What the study of SF seems to suggest is that the intermediate step is semantic in nature, not directly structural in physiological terms, or even strictly functional in the sense of 'a depth of processing effect' whereby "deep, meaningful kinds of information processing leads to more permanent retention than shallow, sensory kinds of processing" (Matlin 1994). So, from observation of the prosodic features of SF's speech during recall, as well as SF's own account of what he is doing, Chase and Ericsson postulate the hierarchical structure of grouping patterns shown below (Figure 9.2). These speech patterns, (pauses, intonation and stress markers), are known indicators of the underlying mental process (Halliday, cited in (Chase) 1981, p. 171).

> There is a great deal of additional evidence that SF uses hierarchical retrieval structures. Probably the most straightforward evidence comes from SF's speech patterns during recall, which almost invariably follow the same pattern. Digit groups are recalled rapidly at a

normal rate of speech (about three digits per sec) with pauses between groups (about 2 sec between groups, on average, with longer pauses when he has difficulty remembering). At the end of hierarchical group, however, there is a falling intonation, generally followed by a longer pause.

(Chase & Ericsson 1981, p. 171)

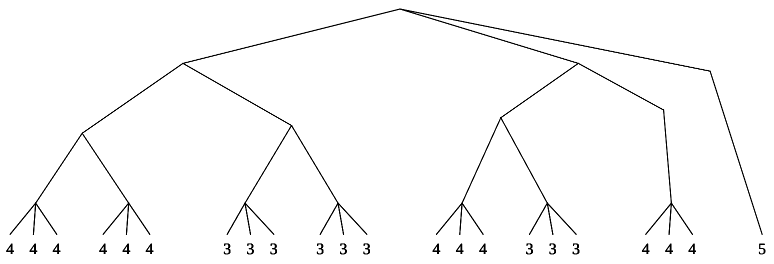


Figure 9.2: Hierarchical organisation of retrieval structure (adapted from (Chase & Ericsson 1981, p. 171)).

All this suggests that what is significant in terms of the performance of the task is the semantic relationships that are being set up, that, in this case, semantics is driving recall. It seems that SF has developed his 'retrieval structure' in long term memory (Chase & Ericsson 1981, p. 177).

If this is true of an essentially meaningless memory task then it is also likely to be a feature of the problem solving process where the conceptual space to be searched for meaning is often enormous. Moreover this fits the contrasting cases of expert and novice performance. The most obvious characteristic of expert behaviour is that experts seem always to know what to do, which aspect of their knowledge to apply, in any particular situation. This is the exact opposite of the novice, whose performance is characterised by *failure to apply* what they are known to know - "it is characteristic of novices that they often fail to apply what they know" (Chase & Ericsson 1981, p. 175). Therefore the most likely explanation of the 'practice effect', the "ubiquitous law of practice" (Newell & Rosenbloom 1981, p. 3), is the development through constant practice of a hierarchical retrieval structure in long term memory. The essential difference between the two levels of performance is thus the utilisation of a semantic structure in long term memory rather than the reliance on cues in short-term memory.

So the truly intriguing question that arises from this, is "can this retrieval structure be consciously set up?" and, the answer, from this study, is that this is exactly what the subject did. He deliberately and systematically built his mnemonic system and retrieval mechanism on an adapted version of a knowledge structure that pre-existed in his mind, and this 'learning curve' is clearly apparent in the graph of his average digit-span as a function of practice over time (Figure 9.3).

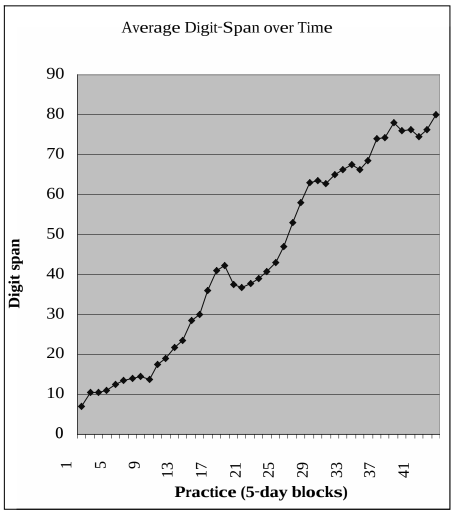


Figure 9.3: Average digit-span over time (adapted from (Chase & Ericsson 1981, p. 144)).

9.4 Pattern Language in this Context

In the discussion so far we have pointed, in passing, to resonances with the pattern language idea, so it is now time to pull the threads of the pattern language argument together. The main difference between the digit-span task and programming is that the programmer is not constrained to working totally in in memory. This means that a lot of the explicit training of memory, the subject's development of his mnemonic system, for example, can be bypassed. Moreover, and indeed, more importantly, there is no need for the adaptation of a pre-existing semantic structure in long term memory, as this can be presented to novice programmers in written form as a pattern language diagram, on the basis that this must have some value in helping them develop one in memory. At the very least, the pattern language given to them should assist novices to achieve more in the way of meaningful practice, and practice can be facilitated further, as we discuss below, by presenting the examples used in the teaching material in the form of a pattern language that is specific to solving the example problem, adding, in effect, at least some of the dynamics of a live example workthrough to the static written form.

An important point that is often overlooked in the pattern literature in computer science is that a pattern language is not just a collection of patterns. It is a diagrammatic representation of the connections between the patterns in terms of their use, the application of one pattern setting up the context in which the patterns below it that are connected to it by arrows are now possible as the next step in the solution. For example, if you are building or designing a wall, having worked through the pattern language in Figure 9.4 to the point of applying WALL the language diagram tells you that you need to think about other things, like doors, and maybe even windows. Thus WALL forms the context in which you can expect to find DOOR and WINDOW. This network of contextual relationships between the patterns in a domain gives the collection the coherence that forms the structure that is known as a pattern language, and that drives the pattern process, just as it is meaningful relationships between words that allow the construction of larger concepts through combination, or the development of productions in the geometry proof example.

The process that derives from the use of a pattern language involves applying the first pattern in the hierarchy, in the case of Figure 9.4, HOUSE, and following the arrows from there. This means here that the designer is prompted to think about the need for a basement. If no basement is envisioned then the designer considers the next arrow from HOUSE. However if such is required then BASEMENT would be applied and the arrows from it investigated in the same way. This procedure of following the arrows continues until such time as a pattern with no further arrows is reached. The designer then backtracks to the point at which the path began, in this case HOUSE. So the pattern language structure is a hierarchy of concepts that drives the design - the list of patterns, the 'pattern sequence' (Alexander 1979, p. 382), that represents the path taken through the diagram is the plan of action for the designer. All that is left to do is to implement that plan, consulting the detail contained in each of the named patterns assisting, as needed, in that implementation.

In examining the journey from novice to expert the Dreyfus brothers found that the way that people cope with the overwhelming proliferation of factors in the problem situation is by adopting "a hierarchical procedure of decision making. By first choosing a plan to organize the situation, and by then examining only the small set of factors that are most important given the chosen plan, a person can both simplify and improve his performance" (Dreyfus & Dreyfus quoted in Taggart (2000)). This is exactly the way that a pattern language works, the solution strategy is directed to the currently pertinent choices by the path taken through the pattern language (a hierarchy). This process of building a plan of the program, a sequence of patterns, through the contextual information encoded in the pattern language is discussed fully in Chapter 7 and an earlier paper (Porter & Calder 2003*b*) through the use of a step-by-step example.

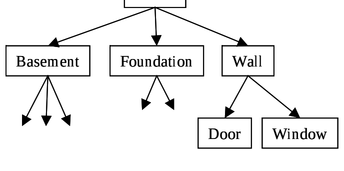


Figure 9.4: Highly simplified portion of a Pattern Language for designing a house.

It is in this sense that a pattern language is a procedural form of the knowledge it represents just as the 'semantic memory' of the expert is. While you are in the planning stage of the solution, all that you want to deal with is the abstract form of the elements needed, not all the messy implementation detail, and this separation of the concept from its detail, the 'declarative form' of the knowledge, is exactly the function that both 'semantic memory' and pattern language diagram enable and empower. But the pattern idea encompasses both forms because while the detailed declarative 'facts' are not present in the pattern language diagram, accessing the detail is simply a matter of matching the pattern name in the collection rather than having to recall them from long-term memory, the expert's recourse, or to search them out in a relatively disorganised set of reference material, the only recourse available to the novice. As the novice's competency increases the need to perform this search decreases but enabling its avoidance at the early stage of her development via the separation of pattern concept from pattern detail is a particularly important advantage that is not available to a novice who does not have the material in pattern language form. In this sense a pattern language is an 'external representation' (ER), and these are known to be important in assisting reasoning.

> A significant proportion of the information in analytical reasoning problems is given implicitly and therefore must be inferred before it can be represented. An important function of ERs is to guide the search for implicit information.
>
> (Cox & Brna 1995, p. 34)

So the critical advantage of 'semantic memory', the network of meaning in the expert's long-term memory, is that it overcomes the 'cognitive load' factor that can be seen to be the main difference between novice and expert performance. Presenting the semantic structure in written form, that is, as a pattern language for introductory programming, bypasses the memory aspect altogether. It might be argued that this adds to the load on short-term memory as the novice has to continually refer back to the pattern language diagram in dealing with the task at hand, but this is not so, because the novice has to do something of this sort anyway with the relatively unstructured information (compared to a pattern language) in the teaching and reference materials. So, if nothing else, the pattern language form is organised for efficient searching and this must have some benefit in terms of cognitive load stress. But, in any case, the main point about presenting the material in this fashion is that, if it is true that the journey from novice to expert is mainly about developing a structured-for-use version of knowledge, as well as the static declarative form, presenting it in that form right from the start should conceivably make the journey easier.

But the pattern language idea doesn't stop there. One of its most significant benefits is the use of a pattern language specific to the problem at hand, a subset of the larger language for domain overall.

> The real work of any process of design lies in this task of *making up*

> *the language* (emphasis added) from which you can later generate the one particular design. You must make the language first, because it is the structure and the content of the language which determine the design.
>
> (Alexander 1979, p. 324)

This can be seen as the functional equivalent of SF's hierarchical retrieval structure, again with the advantage that it is not entirely based in memory as SF's version is. Just as SF develops his recall sequence using the specific retrieval structure that he constructed during the digit readout, so the novice uses the pattern sequence that is developed during the non-coding planning stage based on the use of the overall pattern language diagram to drive the coding (implementation) stage. Note, also, that part of SF's implementation stage, that is the translation of the details held in his retrieval structure to his recall discourse, can be accomplished in the non-coding planning stage in the case of the programming novice. That is, the novice can develop his specific-to-the-task language fully down to a simple sequence of patterns, before she even has to start thinking about code implementation details.

However, the real significance of using a pattern language for introductory programming is that it functions at two levels. That is, as well as facilitating the practice that is essential for the novice to accomplish the journey from the declarative to the combined (declarative plus procedural) form of knowledge through the use of a cut-down specific-to-the-task language, this idea of cutting down the main language to a subset can be used to represent stages in the development of the full language as well, as a series of specific-to-the-stage languages to be presented as appropriate. This means that the journey from novice to expert can be represented in some-knowledge-to-more-knowledge form as well as declarative-to-procedural form. It is a surprise to most people that pattern languages can be as explicitly pedagogical in nature as this, but this overlooks the fact that Alexander's original impetus was pedagogical in intent, if not in setting. In stating that his purpose was to "describe a way of representing design problems which does make them easier to solve" (Alexander 1964, p. 6), he was stating an aim that is pure pedagogy - enabling somebody to perform a task that they were previously incapable of achieving easily.

The idea that explicitly delineating the process of developing expertise in the way described can actually facilitate that development is, possibly, still a contentious point. After all it is clear that even experts fail to appreciate the cognitive basis of their expertise. As Samuel says about the experts in the game that he has consulted in developing his Checkers playing program, "The experts do not know enough about the mental processes involved in playing the game" (quoted in Dreyfus & Dreyfus (2004)) to provide him with the "compiled heuristics" required to bring his program to expert status. Nevertheless it is clear from SF's discussions that he was very conscious of explicitly setting up the structures in memory that he describes and that it was effective in improving his performance. Chase and Erickson make this point in comparing SF's progress to that of another subject who was unable to improve much after the first few days and who quit after a couple of weeks.

> In contrast to SF, this subject never developed a mnemonic system and consequently was never able to improve very much. Notice that the performance of both subjects is very comparable through the first 4 days of the experiment. In fact, on Day 4, SF gave us a fairly lengthy verbal report about how he had reached his limit and no further improvement was possible.
> And then, on Day 5, something very interesting happened. There was a large improvement in SF's digit span (a jump of 4 standard deviations from the day before), and, for the first time, SF began to report the use of a mnemonic aid. From this point on, SF showed a steady increase in his digit span as he developed his mnemonic system and the accompanying control structure.
> (Chase & Ericsson 1981, p. 147)

This suggests that it *is* possible to assist the acquisition of a skill by introspecting on the cognitive processes and structures required, at least to some degree, and this supports Ausubel's contention about the *conscious and purposeful* linking of new knowledge to prior knowledge in his Assimilation Theory of meaningful verbal learning (cited in Zeilik, n.d.). Indeed, it is hard to see how any skill can be learned without some conscious attention to the means by which the underlying cognitive structures or strategies (Gagne 1979, pp. 19–20) are formed. The point about the expert's automatism is that it allows conscious attention to be avoided at the time of use, the "inability to introspect on the application of well-learned procedures" effect (Neves & Anderson 1981, p. 83), but does not preclude introspection at the time of their construction which is something that SF clearly did.

9.5 Bridging the Gap

What a pattern language represents is procedural abstraction, the foundation of expertise, or 'semantic memory' in cognitive parlance. The trouble is that expertise is a complex and hard to discern structure in the mind because we just tend to get on with its use rather than introspecting about what it is. "[An] expert's knowledge is often ill-specified or incomplete because the expert himself doesn't always know exactly what it is he knows about his domain" (Feigenbaum & McCorduck 1983). That is, the significance of knowledge to the expert is its function not its structure, "experts use their knowledge live and rarely have the opportunity to consciously reflect upon what they are doing" (Basque et al. 2004, p. 1),and this is a common stumbling block for educators in trying to explain the use of a concept. Expertise is the almost unconscious use of the web of meaning, 'automatism', but the job of the educator is quite different. One study by Hinds et al. has suggested the difficulty in transferring knowledge and expertise to those less knowledgeable derives from the very characteristics that constitute expertise (Hinds et al. 2001, p. 1232). The teacher's programming expertise, her 'automatism', leads to explanations of the process of deriving a solution that are partial and incomplete, but more importantly, based on complicated logic.

Sheil demonstrates this with a program to find the largest value in a set of positive numbers (See Section 6). The method by which this program is produced is based on the formulation of a loop invariant and proving that invariant by induction, in other words, the teacher's explanation is based on a formal process. But in practice nobody does it that way because the solution is already in their 'semantic memory'. When needed it is just retrieved without actually having to generate it from scratch. Most novices need to be given the `SWAP` pattern when they first come across a situation requiring the swapping of the values held in two variables, it is an explicit node in their pattern language. But it quickly becomes internalised and would disappear from any external representation. This is the point about the evolution of understanding discussed in Section 7.7. But it is a problem in teaching because the novice does not have the internal structure that the teacher does, and teaching is ultimately about helping to build the equivalent structure in the novice's mind. So the idea that giving novices an external representation of the knowledge structure involved in thinking about programming makes up, to some extent, for the lack of the expert's internal model would seem to make sense as one of the major difficulties of teaching with a programming language is that novices tend to concentrate on the surface features of the system rather than the higher level structural representation that the expert uses (Chi et al 1981)

The basis of the pattern language idea, recurring form, is an informal process. So the importance of the pattern language idea is that by bridging the declarative to procedural gap it breaks through the most common stumbling block of novice performance, knowing what to do. It gives the novice a way to proceed, saying, in effect, "well you have done this (applied this pattern), now you need to think about doing one of these things (the patterns pointed to)." But, more significantly, it gives the educator a means to present knowledge in usable form. Examples given in the teaching material are static, in pattern terms they are 'sequences', the product of the journey rather than a map through the concept space. Including the pattern language specific to the example provides the map so that the novice can trace the path through the concept space. That is, it adds at least something of the dynamics of the process of deriving the sequence that forms the design as demonstrated in Section 9.4 and as illustrated in Section 7.2.

Making the process explicit in this way avoids the need to resort to dubious explanations in terms of logic. Logic is the language of computer execution not human communication, and, as such, it is not appropriate in explaining the human process of programming. A significant 'side-effect' of presenting examples in pattern language form is that it encourages the novice to think of the problem in non-programming language terms, that is, to explicitly plan before coding. On its own event, this has advantages in cognitive load terms because of the added load of trying to "deal with decomposition issues in the middle of coding, instead of planning deliberately in advance" (Perkins et al. 1989, p. 257).

So it is the cognitive implications of the pattern language idea, its power in, at first standing in for, and later, helping to set up, the semantic network of relationships and the retrieval automatism of the expert. Moreover, it is in this sense that it addresses the significance of Bloom's taxonomy of cognitive development, that is, that learning takes place in stages, each stage building on the knowledge gained in the previous stage - comprehension requires knowledge, application requires comprehension, analysis requires application, and so on. The difficulty with learning to program is that programming presents a problem in respect of Bloom's progression as it requires high-level cognitive activities such as analysis and synthesis virtually from the start of the learning process. The novice's development "therefore needs to be conducted on a partitioning of the material based on something other than cognitive levels" (Porter & Calder 2003*b*, p. 231).

We believe that the evidence of the studies examined here demonstrates that pattern languages, as a product of a partitioning based on the declarative - procedural split, are the best maps of the journey from novice to expert performance currently available. This is because, although the basic facts about programming, the declarative knowledge captured in the individual patterns, stays the same, the evolution of the procedural form of knowledge is expressed in different pattern languages for the various stages of the individual novice's progress. In this way the developing pattern language reflects the "proceduralization and composition" (Neves & Anderson 1981, p. 82) processes that express the growing expertise of the programmer. This 'evolution' of the pattern language involves two main processes. The first of these is the merging of separate languages into one. It might make sense, for example, to present the variable concept as a small pattern language in its own right, and 'merge' this into the main pattern language appropriately for instance and local versions. This merging process is covered in detail in a paper (Porter et al. 2005) and in Section 7.6 here. But as well as this initial incorporation of the pattern language for variable into the main language its elements - declaration, assignment and initialisation - become so familiar with use that they are no longer required to be present in the language and become absorbed into the pattern for variable. Thus the 'evolution' of a novice's pattern language fits Ausubel's "Assimilation Theory of Meaningful Learning" (Ausubel et al 1978). So just as the pattern idea empowers the fundamental process of design at the individual level, it also drives the pedagogical process towards competency as well.

The cognitive load theory discussed in relation to the difficulty of programming applies twofold in the learning situation, of course, because if working memory capacity is a limiting factor in the task of programming itself, then it must be so in terms of learning. Indeed it is the impact of cognitive load theory on learning that has driven the search for instructional design formats that attempt to minimise the impact of the format on cognitive load so that resources are kept free for where they are really needed, understanding the material (Paas et al 2003). In terms of programming instruction, the largest contribution that is made to successful acquisition by the learner is probably that provided by the use of examples. But there are two ways in which examples can be dealt with in the teaching context, by the provision of worked examples in the teaching material, and by practical work, and the question is, which method is more effective. Surprisingly, some research suggests that studying worked examples is more efficient than solving equivalent problems yourself (Atkinson et al. 2000).

Moreover this effect was found in situations involving worked examples that "showed the begin state of the problem, the goal state, and the solution steps required to reach that goal state" (Van Gog et al 2004, pp 250–1). This would seem to miss out an important aspect of the problem solving process, namely, the strategic thinking behind the steps in solution that must naturally occur in solving the problem, or an equivalent, yourself. Of course that is precisely the factor that proves difficult for novices in solving a problem themselves, so it would seem that the way around the difficulty is to present the strategic thinking required in the apparently more effective format, worked examples. The major hurdle to taking this route is to do it in a way that does not effectively 'blow out' the cognitive load implications of the simple worked example format, because most attempts at presenting strategic thinking in a written format involve lengthy and convoluted explanations. A common approach to presenting material that involves lengthy and convoluted expositions is to resort to a graphical form. What is required of the additional material is for any additional load to be of the germane type, that is, it is effective in learning, not just extraneous to learning (Van Gog et al. 2004, p. 252).

It is a commonplace assumption that both programming and learning are closely related to general thinking skill. If this were not so then the various attempts to measure 'intelligence' in terms of constructing educational systems or in predicting individuals' outcomes in various academic situations would be pointless. The IBM Programmer Aptitude Test (PAT), the Wonderlic Personnel Test, the test of Primary Mental Abilities (PMA) (Mayer & Stalnaker 1968, p. 657), and the Aptitude Assessment Battery Programming (AABP) (DeNelsky & McKee 2005) were early attempts to measure general intellectual ability levels as a predictor of success in learning to program. Typical justifications for this kind of predictive testing run along the following lines:

> An aptitude test attempts to measure whether the applicant has the skills necessary to learn a new set of tasks. Aptitude tests are commonly used for entry-level programming jobs as well as for internal recruitment from other non-programming divisions within an organi-

> zation for the purposes of retraining.
> The Berger Aptitude for Programming Test, Form D (B-APT D) is by far our most popular aptitude test among our clients. This test was designed for measuring the ability of a person to learn the skills necessary to succeed in programming training for the mainframe world. This test is widely used as the first step in admission to programming training in many Fortune 500 companies.
>
> (Psychometrics Inc 2005)

However there are several problems with this type of testing. Firstly the tests typically "measure either a single cognitive factor or a mixture of cognitive factors which have not been separated" (Scanlan 1988, p. 737). Programming involves many different mental activities, therefore it is unlikely that there would be a strong correlation with a single cognitive factor, but equally, an undifferentiated mass of intellectual skills is difficult to relate to programming performance in any meaningful way. Therefore "it is unclear which specific abilities included in these tests relate most strongly to performance" (Webb 1984) in learning to program. But, more importantly, these tests are, more or less, measures of general intelligence (Mayer et al. 1986, p. 608), 'tweaked' IQ tests in effect, so it is difficult to see why they would be more predictive of programming aptitude than of aptitude for any other type of work based on mental activity. As Mayer points out it is difficult to measure the skills required for programming except by teaching it (Mayer 1985).

One of the major difficulties for the learner in constructing the base of knowledge that underlies skill, the programmer's mind, is that the stress of acquisition is accumulative. As we have seen, the expert's skill is based on the use of long term memory rather than the easily overloaded working memory, but the only way of transitioning information into long term memory is by means of organised groups of the 'chunks' used by working memory, rather than the chunks themselves. But if the fact of having to process the chunks in order to handle the immediate programming situation isn't enough to 'overload' the cognitive system, then the additional task of organising the transfer will certainly accomplish that feat. Moreover it has been demonstrated that people who are under any sort of cognitive load will tend to believe false information (Gilbert et al 1993) which implies that they are constructing and transferring incorrect schemes into long term memory. Using this unreliable base in further programming tasks will, of course, cause difficulties that lead to further cognitive stress and the incorporation of more flawed schemata, a typical "vicious circle" type situation. So once the faulty structure has been set in place, correcting it will be difficult, as it is well known that reassessing information is, unlike the bulk of normal cognitive processing, secondary and conscious, rather than primary and at least partly unconscious, that is, it is resource intensive (Gilbert 1991).

All these factors derive, in the end, from the way that the learning process is conducted. This much is made clear by the difference between between novice and experienced programmers. It would appear that much of the difference is due to fact that the novice's strategy is tightly coupled to the programming language while that of the experienced programmer is "often disconnected from a specific language" (Blaschke 2000, p. 9). Closely related to this is the difference between the way that incoming knowledge is encoded, that is, declaratively, and the way that it is used, procedurally. Translating from the declarative form (the set of static facts) to the procedural form (the dynamic skill) is a cognitive task. A large percentage of the difference between expert and novice performance is most likely due to a reduction of the cognitive overhead of this dynamic interpretation of the set of facts, the declarative representation has become transformed into a procedural form so that there is "no costly interpretation phase" (Neves & Anderson 1981, p. 61). This, of course, is the 'practice effect', often stated as 'practice makes perfect', or more correctly "almost always, practice brings improvement, and more practice brings more improvement" (Newell & Rosenbloom 1981, p. 1).

What seems to be happening in this transition is the more efficient organisation of the basic facts of a domain in terms of their use in solving problems in the domain - the expert has, probably, both a greater degree of organisation and a form of organisation that has been honed by practice into a more useful state. One of the main theoretical underpinnings of the idea that "there is no essential difference between practice and learning" (Woodworth 1954, p. 156), is the chunking hypothesis whereby the original primitives, the set of facts learned by the novice become progressively organised into ever larger pieces of knowledge (chunks) which themselves are organised pieces of knowledge (smaller chunks). "A human acquires and organizes knowledge of the environment by forming and storing expressions, called *chunks*, which are structured collections of chunks" (Miller 1956).

All this confirms Dikstra's point, quoted at the beginning of the Introduction that 90% of the teaching of programming is the inculcation of a sense of process, "the teaching of thinking", in fact, not the bare facts of the matter (Dijkstra 1982, p. 1). But we are simply not explicit about the fact that we are teaching problem solving, that is, general thinking skills. Ultimately we are teaching nothing but the logical system itself, the progressive organisation of the "facts" that we teach into the "structured collections of chunks", the "semantic memory" that underlies skilled performance, is left largely to chance. If it performs no other function the pattern language idea does, at least, address the issue of organising the necessary facts into useful form.

Chapter 10

Measuring Pedagogical Effectiveness

We are forced to participate in the games of life before we can possibly learn how to use the options in the rules governing them.

Johann Wolfgang von Goethe

10.1 The Limits of Empiricism

There are several problems in taking too strictly the idea that 'scientific method' is the only path to 'truth', several of which pertain directly to the subject of this project. We have seen, for example, that programming, as an essentially creative activity, is at least as much of an art as it is a science. The scientific base of computing is not in doubt, but the science itself cannot help one solve a problem expressed in some larger system for which an automated procedure is required. What the science provides is the *means* to carry out the task of automation, the programming system, little else, and this is made apparent in the continuous, and continuing, thrust to make programming systems "easier to use". For what else is the fact that they are widely perceived to be difficult to use than an admission that the base level, the scientific core of computing, is not a good fit with the human activity of solving a problem? If it were to be true that the facts of the matter were all that were required then the universal observation (Winslow 1996) that, despite being able to display an adequate grasp of programming knowledge, most students cannot translate this knowledge into programming practice, would simply not pertain.

> Students with a semester or more of instruction often display remarkable naivete about the language that they have been studying and often prove unable to manage dismayingly simple programming problems.
>
> (Perkins et al 1988, p. 154)

But there is an even more fundamental problem because the idea that "progress" in science itself is based purely on empirical methodology is itself an illusion. As we have discussed elsewhere[1], scientific progress depends at least as much on creative leaps into the unkown as it does on empirical research. All that empirical research can do is produce data, and data, without interpretation, is meaningless. Deciding what it means is always a matter of creative imagination. The data concerning the motion of the planets, for example, remained the same over several centuries, yet the understanding derived from it changed radically during the same period, and what changed was the *meaning* derived from the data. So the idea that science concerns nothing but the facts cannot be correct.

> Schools have been impressively successful in spreading the myth that science has a special method of arriving at the truth, that scientific truth is free from value judgements, transcends all cultures, and holds for all time. Any discipline that cannot use the methods of science, the myth holds, cannot establish "objective" knowledge, in short, the discipline cannot establish immutable truths. Given the historical fact that scientific "truth" has changed from the times of Copernicus, Galileo, Kepler, Newton, and Dalton, it seems incredible that the myth of immutable and culture-free science is so persistent.
>
> (Novak 1977, p. 38)

The idea that programming, a *mental* activity, can be defined entirely by the means provided by the programming system, the hard core science, is, effectively, a claim that the human mind, at the level of mental activity involved in programming involves the *same* means - a claim, in fact, that patterns of experience are irrelevant to programming. It may, or may not, be true that, at the *neuronal* level, the human brain parallels in some fashion the operation of a computer[2], but this is clearly not the case at the mental level, otherwise programming a computer would be trivially easy. In terms of much cognitive activity, such as visual perception and the like, the computer model of the brain is clearly a useful analogy, but in terms of any "consciously directed" mental activity we encounter what many cognitive scientists, such as Howard Gardner, themselves see as a "computational paradox".

> The kinds of descriptions that are legitimately offered in the terms of a digital von Neumann computer may turn out to be appropriate accounts of these human cognitive processes. ... But as one

[1]notably, in Chapter 3.

[2]The trouble with trying to understand thinking in terms of some model of the 'real world' is that such models come and go, so one is merely reflecting the latest 'fashionable theory' in physics. Thus in Descartes' day the mind was seen in terms of mechanisms, then came force fields á la Newton and "elective affinities" in the sense of Priestly's valences, and finally the various electrical analogies - such as switchboards in the telephone era and computers these days. Even epiphenomenolism can be taken as reflecting the "parallel universe" aspect of the Many Worlds interpretation of quantum mechanics.

> moves to more complex and belief-tainted processes ... or judgements concerning rival courses of action, the computational model becomes less adequate. Human beings apparently do not approach these tasks in a manner that can be characterized as logical or rational or that entail step-by-step symbolic processing. Rather, they employ heuristics, strategies, biases, images, and other vague and approximate approaches. The kinds of symbol-manipulation models invoked by Newell, Simon, and others in the first generation of cognitivists do not seem adequate for describing such human capacities. ... Human thought emerges as messy, intuitive, subject to subjective representations - not as pure and immaculate calculations.
>
> (Gardner 1985, p. 385)

But Gardner's "computational paradox" is just Karl Popper's *epistemological paradox* restated. What Popper was attempting to do was to "make a clear distinction between the *psychology of knowledge* which deals with empirical facts, and the *logic of knowledge* which is concerned only with logical relations" (Popper 1959, p. 30), in order to avoid the infinite regress that the quest for irreducible facts involves. At some point in the reductionist program the problem of what constitutes observable fact crosses over into the psychological problem of what it is that we perceive. "The central problem of epistemology has always been and still is the problem of the growth of knowledge. *And the growth of knowledge can be studied best by studying the growth of scientific knowledge*" (Popper 1959, p. 15). So the changing nature of scientific *knowledge* demonstrates that it is, fundamentally theoretical, and that the driving force of change is the notion of a "falsifying hypothesis". "The requirement that the falsifying hypothesis must be empirical, and so falsifiable, only means that it must stand in a certain logical relationship to possible basic statements; thus this requirement only concerns the logical form of the hypothesis" (Popper 1959, p. 87).

However, the problem goes even deeper than the misfit between thinking and scientific methodology, there is a misfit at the educational level as well. Like thinking, learning is not a simple cognitive process that can be reduced to anything like "pure and immaculate calculations," it too is "messy, intuitive, [and] subject to subjective representations." In short, it involves the complex mix of meaning and memory, examined in Chapter 9. So although it is possible to agree, in principle, with the proposition that changes to pedagogical method should be based on empirical rather than anecdotal evidence (see (Daniels et al. 2004)), in practice it turns out to be extremely difficult, if not impossible, to establish such an empirical basis for change. There are simply too many factors involved in the learning relationship that make it impossible to control enough of the variables to draw meaningful conclusions.

> What causes schools' mathematics curricula and teaching methodologies to change over time? To what extent do they change in a rational response to external objective considerations; to what extent

> subjectively in accordance with beliefs and social pressures? What does success mean in relation to change? Often enough, the effect of change (planned or otherwise) is to metamorphose antecedent success criteria to validate the change, at least in the short term.
>
> (Macnab 2000)

Moreover there is an important sense in which this is simply a reflection of the way that the human world is - "unlike patterns in life forms, the patterns of social and school structures are changing on a time scale of decades rather than millenia" (Novak 1977, p. 32). When one thinks about it, changes, important changes, in the way that society works are made, or even just happen, without any attempt at trying to assess the ramifications. Sometimes there are attempts to gauge the economic or political implications, but, at best, these are limited in scope and dubious in intent. Often they appear to be more in the way of justification for decisions that have already been made than a real attempt to provide options. What all this rapid flux in social structure means, of course, is that, as a species, we are engaged in a gigantic experiment in mind formation, because mind is the embodiment of human experience, the interface between the brain and reality.

Furthermore, it is a matter of history that the modern field of pedagogy is itself largely a response to changing social circumstance. The modern mass education system arose out of the acceleration in social change that has occurred over the last few hundred years. As the methods of production changed, the ancient way of passing on working skills, the master-apprentice relationship, simply became uneconomic. This was paralleled by other social effects which meant that the 'education' of children became an issue that had to be dealt with on a wider basis than family or church. In itself, the institutionalisation of education can be regarded as a vast social experiment which was begun out of necessity rather than because of any empirically derived imperative. We did not, indeed we *could not*, know in advance what effect the systemisation of childhood experience would have on the development of the human mind, we simply did it. That we now have to live with the results of these vast experiments should come has no surprise. It is just not reasonable to expect that the nature of mind will not reflect changing circumstance.

Aspects of the individual, like expectations, motivations, imagination, and so on, 'forces' in the individual psychological field, so to speak, are bound to be affected by the sort of change that characterises the modern era. The patterns of life are different today from what they were even a decade or two ago, and continue to change apace. One example that is simple in origin, but massively complex in implication, is the advent of modern forms of entertainment. It would not be unreasonable to expect that film and television have changed the way that imagination works. Written works have to be read, the world being presented is 'imagined' by the reader. In other words the act of creation is carried out as much by the reader as the author, one is an active participant in the world that is being created in one's head. This is simply not true of watching film or television, these are sensory rather than imaginative experiences for the viewer so the sense of participation in the act of creation is diminished, if not entirely lost. Viewing is, therefore, a much more mentally passive experience, a play, through the senses, on emotion rather than creativity, and this parallels the other changes in society in the direction of consuming over producing. Is it any wonder that educators find difficulty in stimulating the imagination of their students. The totality of human experience now is vastly more passive than it used to be, much of what we know is given rather than understood, received complete rather than built from scratch.

There is something of a dichotomy here for it is a feature of the history of new technologies to propagate from being used by specialists to general use, that is to change from items of general consumption to items promoting general creativity. Even writing, which we now regard as an almost universal activity, was once the preserve of professional scribes, and similar progressions can be seen in other fields such as photography, computers, and so on (Davis & Moar 2005, p. 158). The drive to make computers 'easier' to use was, and is, driven by the progression from specialist to general use, for example. So maybe the contrasting trends towards wide availability on one hand and passive involvement on the other are conflicting with each other, at least at the level of education. Creativity is an active rather than a passive involvement.

10.2 The Measurement Problem in Education

The attempt to base pedagogical practice on empirical data implies that we know exactly what it is that is being measured by the data. But as we discussed in Section 8.4 it is not clear that we even know how to assess quality in program terms, let alone programming or programmer terms. Ultimately, any assessment of the degree to which someone has assimilated knowledge of some kind is a more a measure of the effectiveness of the pedagogy employed than anything else. If it is not the capacity to *learn* that defines the human condition, then it is difficult to envision what does. Humans display an innate learning capability from day one of their emergence into the world, so if the attempt to *direct* their learning in some direction can be shown to be failing then the failure lies in the *teaching*, not the *learning*. It shows that we have not yet learned how to address the human mind in terms of the subject matter being taught - not, anyway, in a widely applicable sense.

The data generated from educational practice is mostly that derived from the activity we call "assessment". Unfortunately, this data is mostly seen as a way of measuring the individual performance of the learner compared to her peers. But this narrow interpretation is completely misguided in terms of the value of the data to the education process itself. As much as it is used, and is useful, in guiding the assessee's educational and post-education future, this factor is post facto in terms of the situation that generated it. So, while testing and reporting, the giving of marks or grades, is important, assessment is, or should be, more about monitoring how effectively the teaching process contributes to learning. Van De Walle (2004) describes it as having a number of purposes, including:

- Monitoring student progress in order to promote growth
- Evaluating the teaching program so that it can be adjusted
- Making instructional decisions to improve instruction
- Evaluating student achievement so as to recognize accomplishment

What is significant about these purposes is that they are concerned with two agencies - the learning agent and the teaching agent. Knowing where a particular learner stands in relation to the body of knowledge being studied is important mainly in terms of adjusting the study material to better suit their current status - to address any shortcomings in their understanding revealed by the assessment process.

This is not to overlook the fact that there are individual subjective factors, aptitude, motivation, commitment and the like involved. The point is that any educational project *must* engage these factors, and therefore any assessment regime is, to a large extent, merely measuring the degree to which they have been engaged. In the end, the force that drives change in pedagogy derives from the *apprehension* of the fact of failure, not the data that tells us we have failed, because that is all that the data *can* tell us. The response to failure, in other words, is essentially *created.* How to deal with the failure is always going to involve judgement, a creative leap of imagination, because the data reflects the past, not the future. This should not dismay us, it is, in a critical sense, our job as educators to figure out how to educate. And, as with the programming system, the underlying empirical facts of the assessment system, in fact, any system of measurement, for that is what, fundamentally, empiricism is, provide the *means* for creative response to the problem, not the actual solution.

If empirical methodology cannot be useful in terms of actually *creating* a solution, it can, nevertheless, provide some indication of the degree to which the solution works compared to the original situation *after* the solution has been formulated. In this regard the suggestion made by Joe Bergin in the early stages of this research to make an attempt to measure the effectiveness of patterns in the teaching of programming was followed up in three separate experiments, discussed below in Section 10.5. The basic premise here was to attempt to measure the effect on programming performance of different teaching materials. There are some major methodological difficulties with measuring the performance of a skill in this way, so probably the main contribution we made was to test the use of a pattern language as a means of overcoming some of these methodological problems. Probably the main stumbling block in measuring the performance of a skill is the confounding nature of the various factors that are involved. How does one isolate the purely subjective aspects of individual performance?

The answer to this question, suggested by the emphasis of pattern language on process rather than knowledge, is that one doesn't. If the point about pattern languages is correct, then the "whole" of the personality is involved. What the raw knowledge of the *facts* does not necessarily engage, is the sense of becoming, of being involved in the *discovering*, the creation of, knowledge, and this is the fundamental source of the problem. The raw facts, on their own, do not present any sense of process, any notion of the generative forces that they represent and imply. This is why the transition from *knowing* to being able to *apply creatively* is so difficult given lack of appreciation of the connectivity between the facts. What drives the organisation that we see all around us is the web of relationships between things, and, as Spinoza pointed out, "the order and connection of ideas is the same as the order and connection of things" (quoted in (Grabow) 1983, p. 74) .

The difficulty involved in any comparison of different programming pedagogies is that there is no essential difference to be measured if they are simply different ways of presenting the facts that do not engage the sense of process of order implicit in the raw data. Programming is an active use of knowledge, not just simple knowledge of the facts, so two pedagogies that are different only in terms of the facts and not in terms of the process of using the facts are, in essence, not different at all. Historically the big difference between what might be termed "active pedagogy" and "passive pedagogy" has been access to the source of process. Any practical or dynamic use of facts *must* derive from access to such a source, and this was the power of the apprenticeship model. The master, as a dynamic representation of knowledge, is automatically a source of process in a way that symbolic form is not, and the only way to represent any sense of process in static symbolic form is through the relationships between the items involved in the knowledge as demonstrated in the digit span example (see Chapter 9).

So what the pattern language adds to the process of empirically assessing the effectiveness of a pedagogy is the same as it adds to the process of understanding, the means to actively *use* knowledge, not just passively store it in memory. In other words, a pedagogy based on the relationships between the raw facts of the matter is different in *kind* from a pedagogy based on just the facts (Mill 1981, p. 5), and therefore provides the scope for measuring a real difference in effectiveness. Because humans do not approach tasks "in a manner that can be characterized as logical or rational or that entail step-by-step symbolic processing" (Gardner 1985, p. 385), just giving them the symbols alone is not enough. One needs a pedagogy that *enables* process, not one that *assumes* that processing ability is somehow innate and that all that needs to be done to empower it is to provide the symbols and operators. The point is that the symbolic system arises originally from process, not process from the symbolic system, we created the symbolic system in order to do something that we couldn't do as easily without it - explore logical or mathematical relations.

So even where some aspect that is clearly involved in learning, such as the ability to come up with explanations, can be identified as a "relatively stable person characteristic"(Renkl 1997), even this apparently "innate" characteristic seems more to do with the quality of the internal representation of knowledge than a a part of simple individual nature, when examined more closely.

> The individual differences in the quality of self-explanations were, however, found to be multidimensional. Most importantly, even when controlling for time-on-task (quantitative aspect), learning gains could be substantially predicted by qualitative differences of self-explanation characteristics. Successful learners, in particular, tended to employ more principle-based explanations, more explication of operator-goal combinations, and more anticipative reasoning.
>
> (Renkl 1997, p. 1)

Descriptions like, "principle-based" and "operator-goal combinations" are more redolent of the symbol system than base "personality", suggesting again that identifiable aspects of the use of a symbolic system have more to do with the degree to which the system has been "internalised" than anything else.

In this respect it is interesting to note that other versions of the pattern language idea have arisen in other areas of knowledge *from the opposite direction.* That is, whereas Alexander's notion of pattern language is based on empowering the process of design, the "Concept Map" idea arose directly out of an attempt to measure "conceptual understanding", to measure, in effect, the result of the process of learning, the design of the knowledge base in the mind of the learner if you will, rather than to drive it.

10.3 Concept Maps

Thus Concept maps, visual graphs consisting of nodes, which represent concepts, and arcs, which represent relationships between the concepts (Kremer 1997, p. iii), were discovered by means of Piagetian style interviewing (Novak 2004) of students about their knowledge of a subject at various stages of its development. The initial idea of the interview process was to establish the status of a student's knowledge in order to fit instruction to the level of understanding that the student already had. This is in line with Ausubel's Assimilation Theory of meaningful learning (Ausubel et al. 1978).

> The underlying basis of the theory is that meaningful (as opposed to rote) human learning occurs when new knowledge is consciously and purposively linked to an existing framework of prior knowledge in a

> non-arbitrary, substantive fashion. In rote (or memorized) learning, new concepts are added to the learner's framework in an arbitrary and verbatim way, producing a weak and unstable structure that quickly degenerates. The result of meaningful learning is a change in the way individuals experience the world; a conceptual change.
>
> (Zeilik n.d.)

It was found that the information gathered from the tapes of the interviews could be best organised into a hierarchical structure "with more general, more inclusive concepts occupying higher levels in the hierarchy and more specific, less inclusive concepts subsumed under the more general concepts" (Novak 2004, p. 4). This is the form that Ausubel had suggested that meaningful knowledge takes in the learner's mind, and it came to be called a 'concept map'. As it turned out, translating the interview material in this way was both easy and useful, not only in ascertaining the current status of an individual's knowledge but in tracking its development over time.

> We found that a 15-20 page interview transcript could be converted into a one page concept map without losing essential concept and propositional meanings expressed by the interviewee. This we soon realized was a very powerful knowledge representation tool, a tool that would change our research program from this point on.
> In the history of science, there are many examples where the necessity to develop new tools to observe events or objects led to the development of new technologies. For our research program, the necessity to find a better way to represent childrens' conceptual understandings and to be able to observe explicit changes in the concept and propositional structures that construct those meanings led to the development of what has now become a powerful knowledge representation tool useful not only in education but in virtually every sector of human activity.
>
> (Novak 2004, pp. 4–5)

The particular context for these developments was a twelve year, longitudinal study of a two year cohort of students as they progressed through school, starting from when they were in first grade (6-8 years old). One half of the two year cohort, that is the first grade students in the first of the two calendar years, was given special instruction in basic science concepts concerning the nature of matter and energy, while the those doing grade one in the second of the two years acted as the control group, receiving no such instruction. Assessing the impact of this instruction on students would be the task performed by the concept maps derived from discussion with the students.

> Twenty-eight lessons were developed that dealt with the particulate nature of matter, energy types and energy transformations, energy utilization in living things, and other related ideas. For the most

> part, these kinds of concepts are rarely presented to elementary school children, especially to 6-8 year olds in grades one and two.
>
> (Novak 2004, p. 2)

As it is not normal for children to be introduced to material of this kind at such an early stage, the idea was to measure any effect on their assimilation of these concepts when they encountered molecular kinetics and energy transformations later in their schooling. Instructional material was specially developed to match the level of the student's understanding on the basis of the principle espoused by Ausubel:

> If I had to reduce all of educational psychology to just one principle, I would say this: The most important single factor influencing learning is what the learner already knows. Ascertain this and teach him accordingly.
>
> (Ausubel 1968, p. vi)

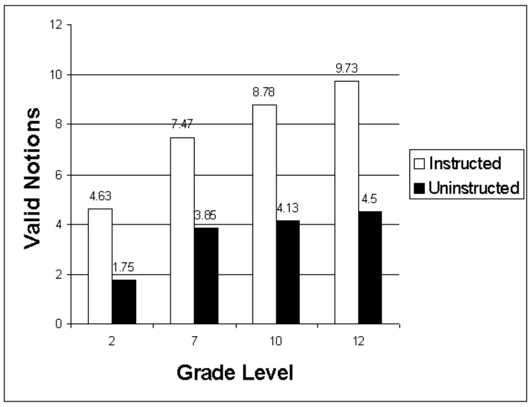
What the concept maps provided was a way of comparing an individual's understanding with both the model being taught and with other student's understanding, but, perhaps more cogently, with their own understanding at different stages. "The precision and clarity of the learner's cognitive structure represented this way made it relatively easy to follow specific changes in the student's knowledge structures as she/he progressed through the grades" (Novak 2004, p. 5). Moreover, as the project progressed, other studies with a different set of students (in order not to confound the main longitudinal study) were conducted in order to gauge the potential for students to construct their own concept maps "by giving them key terms which they had to arrange in meaningful patterns and then connect with lines that they labeled with the nature of the relation between the terms" (Novak 2004, p. 6), and these were found to be equally informative in illuminating the student's cognitive structures.

By looking for both valid and invalid notions in the student's understanding, it was possible to measure the fit with the correct model, enabling the population and temporal comparisons to be made.

> It was clearly evident that Instructed children had fewer and fewer misconceptions as they progressed through school, when compared with Uninstructed students. Conversely, the Instructed students had an increasing number of valid ideas or notions as they progressed through the grades. The results are shown in Figure 10.1. We see that by the end of grade 2 the Instructed students significantly outperformed the Uninstructed students in their understanding of energy and molecular kinetics ideas. When students begin the formal study of science in grade 7, both Instructed and Uninstructed students improve in their understanding of energy and molecular kinetics concepts, but a highly significant ($p < .001$) superiority of Instructed students compared with Uninstructed students was observed, both for valid and

invalid ideas. Moreover, the Instructed students showed steady improvement as they progressed through high school science courses, whereas improvements for Uninstructed students were small. This significant difference in performance over the years for the Instructed and Uninstructed groups led to a significant interaction variance for years in school. Other statistical results have been reported elsewhere (Novak and Musonda, 1991).

(Novak 2004, p. 7)



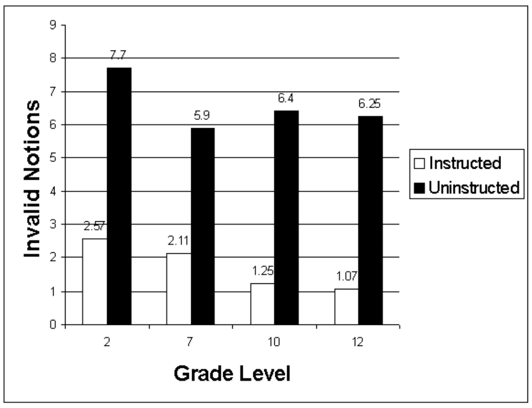


Figure 10.1: The number of valid and invalid notions held by Instructed and Uninstructed students in grades 2, 7, 10, and 12. (adapted from (Novak 2004, p. 7)).

Because the trial began with a large number of students - 191 received the special instruction in the first year - the attrition rate over the 12 year period was not fatal to the project. Out of the more than 300 children in both groups, 87 remained available to the end, 85 of whom were interviewed in the final year. Moreover both the instructed and the control group achieved virtually identical Scholastic Assessment Test (SAT) scores, suggesting that the samples were comparable in general intellectual ability. What is significant in this experiment is the advanced nature of the concepts introduced in relation to the age of the children and the fact that the hierarchical organisation of the information garnered from the interviews proved to be the best way to provide a measure of comparison of levels of understanding. Various forms of tests were tried initially but were found not to be "valid indicators of the conceptual understanding of students" (Novak 2004, p. 3) that could be demonstrated in the interview process by individuals.

The significance of the concept map is therefore in its 'objectification' of the subjective state of the student's mind in terms of the particular set of concepts involved, overcoming the difficulties involved in observing changes in cognitive structure over time (Novak 2004, p. 4). But what this means is that the concept map notion addresses the ubiquitous problem of knowledge transfer (Hinds et al 2001) , because the difficulty that is involved is the fundamentally the same, communicating understanding. The person producing the concept map is trying to communicate the understanding demonstrated by the student under questioning to others who read the report of the interview, while teachers are attempting to communicate their own understanding of the teaching material to their pupils. An example (Figure 10.2) of a concept map drawn from an interview with an above average instructed student at the end of grade 2 is presented here. A version from the same student at the end of grade 12 is presented in (Novak 2004, p. 5), but the main issue here is to demonstrate the essential similarity between a concept map and Alexander's pattern language.

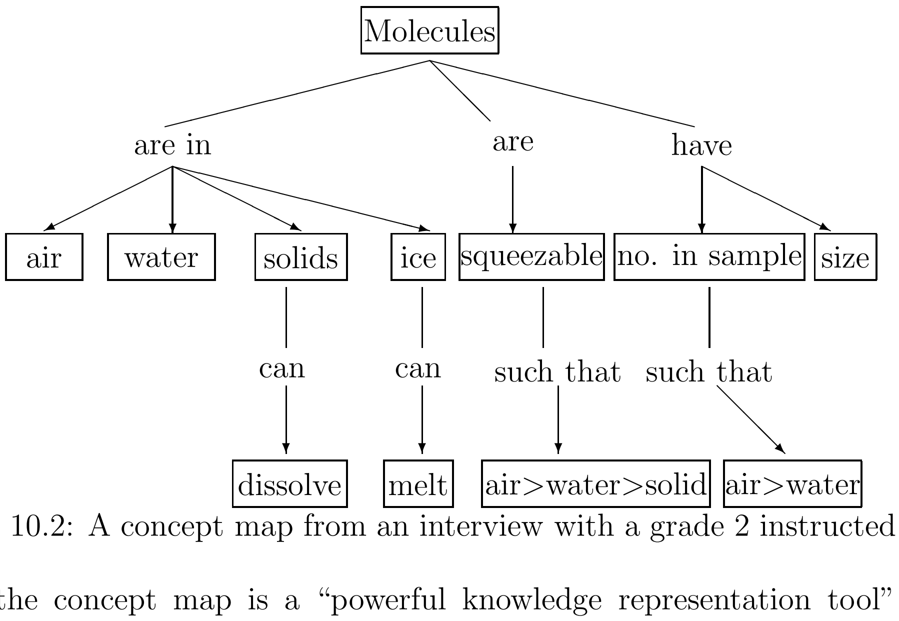


Figure 10.2: A concept map from an interview with a grade 2 instructed student

If the concept map is a "powerful knowledge representation tool" (Novak 2004, p. 4) in enabling the process of assessing the level of understanding of students then presumably it is equally powerful in the process of communicating understanding *to* students in the first place, and Novak, in fact, claims this.

> Another use of concept maps is to provide maps made by experts to serve to scaffold learning of students (ODonnell, Dansereau & Hall. 2002). The idea of scaffolding learning goes back to early studies by

> Vygotsky where he described his studies showing that language and the social exchange using language can significantly enhance childrens cognitive development. Through proper use of language, adults can scaffold the learning of concepts by children.
>
> (Novak 2004, p. 8)

This is, of course, the same notion that underlies the use of pattern languages in assisting people to learn to program. So the question is, 'how similar are these two structures?' Comparing Figure 10.2 to Figure 7.2 clearly demonstrates that we are dealing with the same basic conceptual structure. Concepts are arranged hierarchically "with more general, more inclusive concepts occupying higher levels in the hierarchy and more specific, less inclusive concepts subsumed under the more general concepts" (Novak 2004, p. 4). The most significant difference between the two forms is undoubtably the naming of the links between the concepts in the concept map version. So, for example, the link between MOLECULES and AIR in Figure 10.2, is labeled as *are in* in the concept map, while that between MOLECULES and SIZE has the notation *have*. However both these connections can be seen as specific forms of the usual implied meaning of the connections between patterns in a pattern language, that is, *refines*, but at a finer level of understanding.

In fact the versions of pattern languages produced by some practitioners do explicitly name the links, for example as *uses*, *may use*, *variant* and the like (Kodituwakku & Bertok 2003, p. 65). But even in the more usual pattern language form the pattern for the concept AIR in the concept graph illustrated here might incorporate the label on the link from MOLECULES becoming ARE-IN-AIR, for instance. The main reason for not giving labels to links in pattern languages for programming is that the process being driven by the language is quite specific, that is, designing a program, so, in a sense, the meaning of a link between patterns is always *consider doing this next*, especially so in a language designed to help people learn to program.

So the main difference between a concept map and a pattern language can be identified as the specificity of purpose. It seems that a concept map, because it does not have a very specific purpose, *needs* the links to be specified because they lack the common purpose assigned to links in a pattern language by the overall purpose of the pattern language. The purpose of a concept map is to express *understanding*, a very general notion, while that of a pattern language is to express a specific *design process*. Although both purposes are ultimately about the same thing, conceptual understanding, the understanding in the latter case is being expressed in the programming task and is therefore inherently more specific. It is understanding directed at the task of producing a program rather than understanding just for the sake of it.

But at their most basic level, the very point about pattern languages is that they are based on Ausubel's principle insofar that they relate the unknown to what is already known. A pattern is such because it is a common experience, but its commonness, by itself, does not necessarily make it useful in terms of understanding. Sunrises are common experiences but, of themselves, they do not make for a correct understanding of the solar system - an earth-centred view of the system is possible, indeed likely, if the relationship between many, maybe less noticed but equally common, events are not perceived. It is only when the sunrise pattern is related to the other patterns in the sky (planetary motions) that the true picture about the functioning of the system begins to emerge. In other words, it is the pattern language that gives rise to the *process* of understanding, not the single pattern. The single pattern contains information about the concept, in this case the movement of the sun relative to the Earth, not the connection between concepts that constitute system-level order. The *system* is illuminated by correlating many of the various movements that commonly occur in the sky, that is through the connections between concepts.

Perhaps the most interesting aspect of the concept map idea is the way in which they were 'discovered'. The problem that led to their formulation was that of representing the conceptual understanding of individual students in a form that made comparisons between them possible. Several studies of concept maps as devices for measuring conceptual structure "suggest that the technique has many of the desirable characteristics that testing and measurement experts look for in new assessment tools ... [reflecting] essentially the same structure as that seen in much more time-consuming techniques, such as interviews and picture sorting tasks" (Zeilik n.d.). More usual ways of getting people to express their understanding are either ineffective (Novak 2004, p. 3) or make comparisons difficult because of their discursive and static nature. Interviewing people avoids the second of these last two difficulties, the interviewer can actively probe the student, but a simple transcription of the interview is still discursive in form. Translating the interview transcript into visual and thereby more easily comparable form - or better still, getting the assessee to generate their own version - is therefore the essential task performed by the concept map.

Interestingly this is exactly what the subject of the experiment examined in Section 9.3 did in relation to the performance of the digit span task. But even more fascinating is that the task being performed bore no real relationship to the conceptual structure expressed in the concept map or pattern language used by the subject. This points to the absolute primacy of process in the structuring of information in these ways. In SF's case the phrase "procedural knowledge" is almost an oxymoron because the knowledge being used to drive the procedure has virtually nothing to do with the procedure. The lack of a direct connection between the *knowledge* and the procedure being driven by the knowledge in this case, illustrates the pure generative power of conceptual structure, the fundamental premise of both the pattern language and concept map ideas. What SF was doing was providing the information involved in the task with a context that it did not possess in its own right, giving it *meaning*, in other words. And this is precisely what a pattern language does for a symbolic logic system. The constructs that make up a programming system have no *context* in terms of the real world where the problems being addressed by the programmer actually exist, they carry no *meaning* in these terms, and therefore have no generative power in the sense of *creating* a solution. Pattern languages set each of the programming system constructs in relationship to each other, give the symbolic logic *context*, and therefore meaning, in the real world that it does not inherently possess. This is what happens in the mind of a competent programmer anyway, but, unlike SF it is occurring unconsciously, unless, of course, she is pattern aware.

10.4 Objectifying Procedural Knowledge

Empirical quantitative analysis of the use of concept maps as a means of assessing the knowledge structure of students suggest that it has many advantages in terms of providing relatively objective metrics. But this means that they do more than just address the measurement problem itself, because it indicates that a concept map is a genuine feature of the way that knowledge is structured in the mind.

> Results across all the studies using the construct-a-map technique suggest the following good news about concept map scores: (1) Students can be trained to construct concept maps in a short period of time with limited practice. (2) Raters do not introduce error variability into the scores. Concept maps can be reliably scored even when complex judgments such as quality of proposition are required (the interrater reliability on convergence score averaged across studies is .96). (3) Sampling variability from one random sample of concepts to another provides equivalent map scores when the concept domain is carefully specified. It is possible that the procedure we have followed in selecting the concept domain helped to create a list of cohesive concepts, therefore, any combination of concepts could provide critical information about students knowledge about a topic. (4) The high magnitude of relative (.91) and absolute (.91) coefficients, averaged across types of scores and studies, suggest that concept maps scores can consistently rank students relative to one another and provide a good estimate of a students level of performance, independently of how well their classmates performed. (5) The proportion of valid propositions in the students map out of the possible propositions in a criterion map seems to better reflect systematic differences in students connected understanding than other scores and it is the most effort and time efficient indicator. Other procedures have been carried out for supporting score interpretations (e.g., comparison between experts and novices scores).
>
> (Ruiz-Primo 2004, p. 5)

The concept map in this context is therefore an effective way of translating the state of knowledge in a student's mind into assessable form. But this is exactly the same process that a pattern language addresses, translating domain knowledge, in this case the expert's rather than that of the person being assessed, into a more *useable* form, where useability is defined in terms of design rather than comparison. Given this convergence on the task of 'translation of knowledge into useable form' it should come as no surprise that concept maps and pattern languages are essentially the same. Therefore, as most of the difficulty with the programming task for novices, especially, is exactly this 'translation of knowledge into useable form', proceduralising it, suggests that the effectiveness of concept maps in the task of assessing conceptual structure is a pointer, at the very least, to the efficacy of pattern languages in building procedural conceptual structure.

Unlike other forms of assessing the degree to which a particular pedagogy has empowered the assimilation of knowledge by an individual, concept maps and pattern languages provide a means of assessing the necessary procedural form rather than the static form of knowledge. In a sense, any attempt to measure the performance of a skill like programming is, at base level, an attempt to measure the procedural knowledge of the programmer. The claim made for concept mapping is that it, too, empowers a process, the process of explaining one's understanding. Most forms of measuring the extent of knowledge acquisition, tests and examinations, measure just knowledge, not understanding, static concept level information rather than dynamic system or domain level connections.

By comparing the performance of two groups, one with pattern language instruction and one with more usual instruction what we were really assessing was the degree of *proceduralisation* of the knowledge in each group. So the claim that pattern languages and concept maps empower the assessment of understanding rests on a claim that these forms of knowledge representation drive process, and it is important to discover how this dynamic is derived from what is, after all, still a static representation of knowledge. Fortunately, there is another system that can be seen as a pattern language or concept map, that illustrates this empowerment at work, and this is the system used in identifying biological specimens known as "keying".

> Identification assumes that the plants have already been classified and named. When you identify a plant, you are basically asking: "Of all known species, which one most closely resembles this individual in my hand?"
> Professionals and serious amateurs identify plants by keying. This is a stepwise process of elimination that uses a series of paired contrasting statements, known as a dichotomous key. Keying is like a trip down a repeatedly forking road: If at the first fork you turn right, you cannot possibly reach any of the towns that lie along the left fork. Each successive fork in the road eliminates other towns, until you finally reach your destination.

> When keying, the user begins by reading the first pair of statements (called a couplet). For example, a key may begin by asking the user to decide between "plants woody" and "plants not woody" If the unknown is woody, all nonwoody species are immediately eliminated from consideration. Successive couplets will eliminate further possibilities until only one remains, which is the species to which the unknown must belong. The advantage of this procedure is that the user must only make one decision at a time, rather than mentally juggling long lists of features of many possible candidates.
>
> (Lammers 2005)

Here we can detect the relationship between the details, the form of their presentation, and the process in which they are being used. Although the information required for keying specimens is not usually presented graphically, this is probably due to an extraneous factor, the massive amount of information to be presented - a representation of only 10 levels would have 2^{10}, or 1,024 nodes in the final level. Nevertheless the fact that the presentation is in the form of a hierarchy or tree is made clear by the reference to "forking roads", even though the tree is almost never drawn. The connection between information, form and process is made abundantly clear in this case because the way that the linear process of identification derives from interrogation of a hierarchy of knowledge is simplified by the paired nature of the connections. So, unless an actual wrong turn has been made, the strictly linear nature of the process is always apparent. There is none of the multiple possible routes that occurs in design processes, nor any recursive diversions.

This is as clear an exposition as is possible. The process is entirely derived by making decisions based on the connectivity of information. Concept maps and pattern languages *must* derive their procedural power in exactly the same manner even though the structure involves more complex and recursive connectivity. Clear too, is the nature of the connecting links, because, as with the pattern language case, the links always mean "refines" - refines the identification rather than refines the design as in a pattern language for design. And, again, the clarity is the result of the fact that the process being driven by the pattern language, the process of identifying specimens, is considerably less general than the notion of an "understanding" process.

10.5 Testing the Pattern Process

Following the suggestion by Joe Bergin we decided to attempt to empirically test the effectiveness of an introduction to patterns and a pattern language in assisting the development of programming skill. This decision evolved over time into three separate trials, driven mainly by the shortcomings in methodology that were exposed during each trial. We always expected the first attempt to be something of a trial run for further experiments, but the progression evolved quite differently from what we expected initially.

The situation at Flinders University in regard to the teaching of programming starts with a first year course in Java, Computer Programming 1 (CP1), which is followed up in second year with two more advanced topics, CP2A which is based, like CP1, on general Java programming, and CP2B, which is built around a library of abstract data type material, the ADS package. Whatever methodology that we decided to use, it had to be built around the structure of the teaching program.

The idea of an experiment to test the use of different teaching materials can be implemented in two ways, the most obvious, and optimal, solution being to divide the existing first course cohort into two groups and teach each group on the basis of the different materials being tested. This method ensures a thorough introduction to the material for the participants, and provides, therefore, the most rigorous test, but the logistics of running a test on this scale, not the mention the ethical implications, border on nightmare proportions. It would mean a virtual doubling of the staff effort with the attendant timetabling complications and the problem of keeping the two streams separate.

So a more realistic attempt would have to be built around the existing course structure to take advantage of the state of knowledge expected of students at each stage. By and large, the main thrust of CP1 is to act as an introduction to Java. This can be seen by the fact that 60% of the assessment is based on an open book multiple-choice exam and 10% on multiple-choice quizzes during the weekly tutorial sessions. Therefore only the remaining 30% is based on actual programming performance. Moreover this programming component is done during practical classes where demonstrators and other forms of assistance are readily available.

The second general programming course, CP2A, makes more of an attempt to cultivate programming skills, the assessment regime being based almost entirely on programming performance as such, with 70% being in the form of online exam-condition programming tests. Given this natural difference in the knowledge base being addressed in each course, the conceptual gap between them is the most obvious point at which to run a program that introduces different groups to different materials to test the effect on their programming performance In considering the conduct of an experiment to test the use of different materials we needed to take account of two main factors, what the experimental subjects already know and what they need to do with the knowledge. Their assessment result in CP1 should give an indication of the state of their knowledge base, what they already know, so all that is needed is a way to measure their performance in using the knowledge to write programs.

10.5.1 The First Trial

In the first semester of 2003 we set up the trial to be run as a standalone experiment separate from the teaching program of both topics, the only point of contact being to call for volunteers from the students beginning CP2A. To test the process that arises from using a pattern language we proposed to give two groups of students access to different teaching materials. Both groups would then attempt a common programming assignment under the same conditions so that the effectiveness of the two sets of materials can be compared. One group would be introduced to a set of patterns and an associated pattern language, while the other group would be given material that covers the same ground but is written in a non-pattern form.

Since the patterns group would need both the patterns themselves, and the pattern language diagram that illustrates the contextual relationships between them, we also proposed to test the use of the pattern language idea itself. In surveying the situation in Computer Science it is pretty clear that, by and large, patterns have been adopted in isolation from pattern languages. We can find very few actual pattern languages in the field. This means that the main benefit that Alexander saw in his pattern work, the generative power of the language is missing.

> As in the case of natural languages, the pattern language is generative. It not only tells us the rules of arrangement, but shows us how to construct arrangements - as many as we want - which satisfy the rules.
>
> (Alexander 1979, p. 186)

Therefore by arranging a third group, who would be given the same patterns as the first group, but not the pattern language, we could effectively test this notion of the generative power of the pattern language. This widens the usefulness of the experiment beyond the educational domain into a study of the use of patterns in software development generally.

As Alexander himself has said, while addressing a gathering of several thousand software development people at the OOPSLA conference in 1996, it would appear that in transferring the pattern language idea into computer science the generative and moral aspects that were primary for his own work have been missed (Alexander 1999). If Alexander is correct, the widening of the experiment would demonstrate this in practice. The group with both the patterns and the pattern language should perform better in a programming test than the patterns-only group because of this generative power of the language.

Dealing with people who had completed the first course, CP1, meant that we could use the first course results to control for various degrees of familiarity with the programming language. What we want to test is the effect of new material on programming skill, so having a measure of each person's knowledge of Java provides an important control given the complexity of the connection between knowledge and skill.

Another consideration is that while the material is not entirely new, there is nevertheless, a significant amount of learning required to assimilate the new format and enable its use in solving programming problems. So as well as the pattern material there would need to be a process of introducing it. We decided to provide an introductory lecture and a follow up discussion based on stepping through a couple of examples. The second session would be driven by student participation.

The use of patterns is not entirely novel to these people as the first course is to some extent built around a collection of patterns, but not organised as a language. This project follows on from an investigation of patterns in learning to program at the Honours level, and the current teaching program in CP1 had been developed, in part, from that project. The patterns were especially developed for the course, and were built on material derived from the patterns in programming pedagogy available via various sources on the Internet, but in particular, that on, and linked to, the primary site devoted to the use of patterns in programming instruction, that of Joe Bergin (Bergin 2005), and on those in the book chosen as the text for the course, "The Object of Java", by David D. Riley. It should be noted that although patterns are used in CP1, they are not organised in pattern language form, and therefore the volunteers would have no experience in the pattern process which is being tested.

In order to keep the experimental group experiences similar, introductory sessions for the other two groups would also be needed. Doing this for the group that has no pattern exposure is not too difficult. However the group with the patterns but no language would have to be carefully dealt with, in order not to introduce the pattern language inadvertently in explaining the use of the patterns.

In developing the pattern language for the experiment we found that some of the concepts that we felt needed to be included as patterns did not fit very well into the structure of relationships that derives from the basic syntax of Java. The higher level thinking involved in developing the relationship between classes, in particular, seemed to bear little relationship to that involved in designing the detail of a class. This led us to separate the patterns into two pattern languages, one for designing the program in terms of the relationship between classes, and the other for designing class detail. The use of the first language, which was called "Objects Everywhere" (see Figure 6.4), results in a diagrammatic representation of the class structure of a program. The second language, "Class as Blueprint" (see Figure 10.3), gives rise to a sequence of patterns that specifies the design of a class. (Note: Two examples of patterns from each of these languages are included in Appendix A).

Because we were running the trial on the basis of volunteers, and no inducement other than a possible advantage in acquiring the skill of programming was offered, we expected that the idea would appeal mainly to those who were already

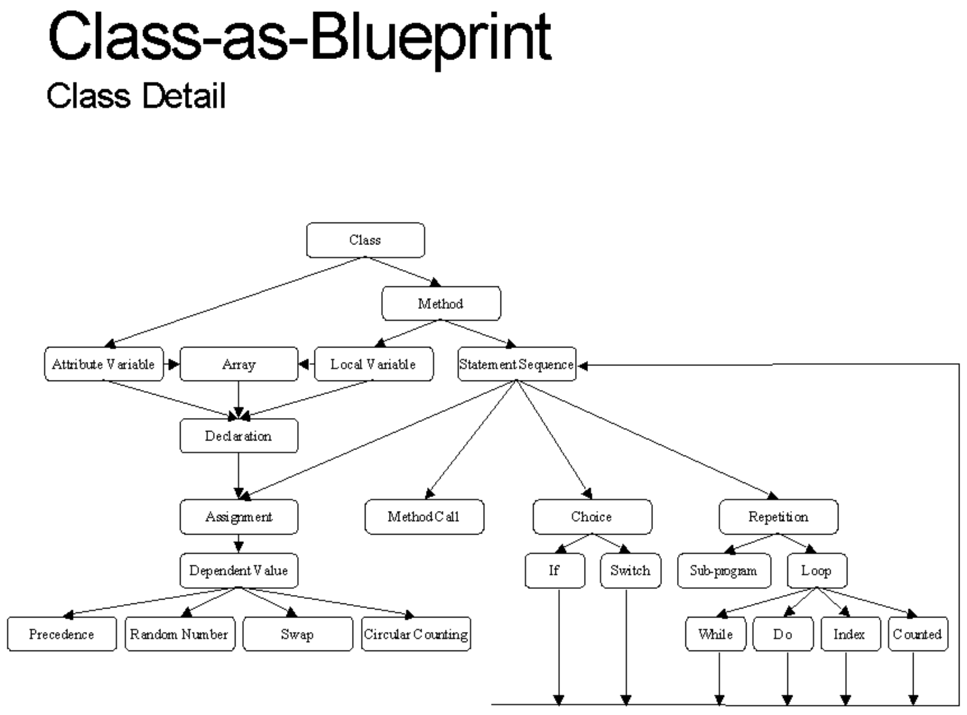


Figure 10.3. The Pattern Language for Class detail.

aware that they were struggling with programming. Of the approximately 300 students starting the second course, 84 signed up as being interested in taking part in the experiment. As expected, this number fell off as further steps in the process were taken, so that by the time the trial began 27 remained.

As we felt that this was not a large enough number to run three groups it was decided at this stage to drop the patterns-only group. So the pool of volunteers was broken down into two smaller groups, a test group of 13, Group A, and a control group of 14, Group B. The test group was given a set of patterns based on the syntax of Java and a pattern language structure in the form of a diagram. The control group was provided with material describing the same basic programming language details that are covered in the patterns, but structured in non-pattern form. This material was loosely based on the course material provided to first programming course students from 1998 to 2001.

The groups were arranged on the basis of the results of the participants in the first programming course, so that each group contained a similar spread of ability. Some people did not turn up for their introductory sessions, so the participation rate was further reduced to 18. The introductory lectures for each group were based around a discussion of the material and an analysis of the process of using it to solve a simple programming problem. Another simple programming problem formed the basis for the workshop sessions and these were driven by student participation.

A week later both groups completed a programming assignment. The programming assignment was taken on an individual basis under the conditions that normally apply for on-line exams in our school. The program code produced was later assessed for the degree of completion reached and on the basis of its quality in terms of programming technique and style.

The programming task set was divided into six stages of progressive difficulty, designed so that anybody who had passed the first programming course should have little trouble achieving stage 1, and that the increasing difficulty of the subsequent stages would cause a spread of achievement that could be used to measure the effect of the different materials. For the control group, we anticipated a correlation between the first course result and the stage reached something like that shown in Table 10.1. Any beneficial effect based on the effectiveness of different materials should show up by shifting the achieved ranges upwards.

Topic1 result	Expected spread of achievement
HD	stage 5 or 6
DN	stage 4 or 5
CR	stage 3 or 4
P	stage 1 or 2

Table 10.1. Expected range for the control group

Table 10.2 summarises the results achieved in the test by the individual participants. The number in the result column is the stage reached within three hours from the start time. Coding style quality is assessed in column 5. The participant ID was assigned randomly in order to disguise the real identity of the students, and the starting letter identifies the group to which the participant was assigned - A for the pattern group, and B for the control group.

ID	Topic1 result	Stage reached	Style
A1	P	2	satisfactory
A2	CR	5	very good
A3	HD	6(almost)	excellent
A4	CR	4	good
A5	P	1	satisfactory
A6	DN	5(with a bug)	good
B1	HD	5	very good
B2	CR	4	good
B3	CR	1	good
B4	P	2	good

Table 10.2. Summary of individual results

Figure 10.4 plots the results achieved in the test, using circles for members of the control group and crosses for pattern users, against the range expected from CP1 performance. The expected range is indicated by lines.

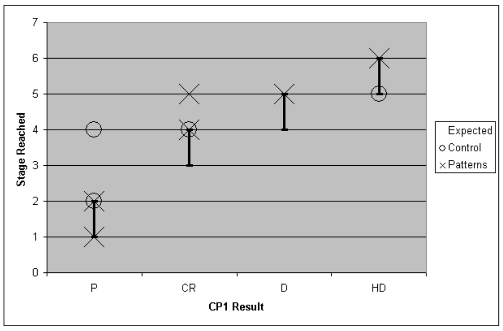


Figure 10.4. Comparison of results with expected range.

What Figure 10.4 shows is that the individual results for the control group matched the expected performance except for one student who achieved stage 4, better than the expected stage 1 or 2. The results for the pattern group tended to be in the high end of the range expected, except for one student who achieved a stage 1 result. Some variation in this first programming course result level is not unexpected because the Pass grade covers a larger range of marks than the other grades.

The number of students who completed the programming task is probably too small to allow any statistically significant interpretation of differences in achievement levels between the two groups. Nevertheless the results do seem to indicate an improved performance for the group exposed to the pattern language as compared to those without exposure. Moreover the exercise was highly useful in providing indications of how the study should proceed, and we discuss this further in the next section.

We feel that the results of the trial indicate that it *is* possible to measure the effects of different materials on performance in this way, but that any subsequent experiment will need to find a way to increase participation levels. Although 84 people from the 2003 cohort (approximately 300) expressed an interest in taking part in the experiment, difficulties in arranging suitable times quickly reduced this to 27. Eighteen people attended the introductory sessions, but only half of these made it to the programming test.

And somewhat in contradiction to the need to increase the active participation rate is the fact that the experiment indicates that the approach of merely introducing people to the materials and expecting them to be used in a programming test is not realistic. That is, there is both a need to increase the length of the time that volunteers commit to the study, and a need to find a way of increasing the number of volunteers completing. Unfortunately, the obvious solution to the participation rate issue, offering payment for participation, conflicts with the commitment issue by complicating motivation.

Probably the most significant finding of the trial in pedagogical terms was that concerning the structure of the first programming course. If we believe that designing a solution before coding is desirable then this needs to be made a primary focus of the first programming course. The trial run demonstrated that introducing the concept of design-before-code is not something that can be done after the learning of the programming language. We need to be clearer about what the aims of the first course are, so that we can be sure that they are addressed in the course material.

10.5.2 The Second Trial

The major shortcoming of the first trial was that the number of volunteers involved was too small to be able to make any statistical inferences. Part of the problem is that having a control group reduces the number of volunteers who are effectively being exposed to the methodology being tested. Therefore for a second attempt we decided to try a method that removed the necessity for the use of a control group by utilising a comparison between CP1 and CP2A results as a means of measuring any difference in programming performance, rather than an explicit programming test. The group exposed to the pattern material is compared to the rest of the class. In this way any effect caused by the exposure to pattern instruction should show up as a difference in the CP1-CP2A correlation between those in the class who received it and those who didn't.

This leaves the question of how the exposure to the pattern material is to be administered and the experience of the first trial suggested that a couple of specially run introductory sessions were probably not sufficient. Accordingly we felt that the best way to conduct this version of the experiment was to offer those in the whole CP2A class who felt that they required extra programming assistance the opportunity to attend a series of special help sessions during the semester. Their performance in the assessment for the topic would then be correlated with their performance in CP1 and compared against a similar correlation of the results of a group of students who had not attended the extra sessions. Because of the difference in the assessment regimes for CP1 and CP2A, comparing the two results correlates the knowledge of the programming language indicated by the CP1 result with the actual performance of programming skill indicated by the CP2A results. Interposing an exposure to pattern-based teaching material between the two topic assessments is thereby a way of assessing its impact on the acquisition of programming ability.

Ethics approval for the use of student's topic results was obtained, and the project was introduced to the students at a normal CP2A lecture. Of the whole class, 85 people signed consent forms for their CP1 and CP2A results to be accessed, and during the semester, 17 attended at least one of the sessions which ran over 10 weeks. The attendance pattern is given in Table 10.3 below. This meant that the pattern group could, theoretically, number up to 17 and the control up to 68, but this would be dependent on results being available for both topics.

	Week	1	2	3	4	5	6	7	8	9	10	Totals
ID												
P1					X	X						2
P2				X								1
P3				X								1
P4				X		X		X				3
P5		X										1
P6				X								1
P7		X										1
P8		X	X	X	X		X	X	X	X		8
P9										X	X	2
P10					X	X	X					3
P11					X				X			2
P12		X			X	X						3
P13					X							1
P14					X				X	X	X	4
P15		X			X							2
P16		X										1
P17					X							1
												37
Totals		6	1	5	9	4	2	2	3	3	2	37

Table 10.3. Attendance at Help Sessions

In practice it was found to be difficult to concentrate exclusively on the pattern material in the help sessions as students, understandably, tended to raise issues of immediate concern to their current assessment task, but the discussion was based as much as possible on the pattern material handed to students at their first attendance. As is usual in extra sessions of this kind attendance tended to be irregular with only 9 people attending two or more sessions. In general, participation in the discussion was quite strong, with virtually all participants contributing to some degree.

For various reasons not all of the volunteers have, at the time of writing, results available for both of the topics being correlated, with the result that the final comparison is between a group of 14 participants and 46 non-participants. Of the non-participants, two were found to have results for CP2A below 10%, which probably indicates that they had not attempted the on-line exam component. These two students were therefore removed from consideration as their result in CP2A is considered meaningless in terms of programming skill and just tends to distort the figures for the control group.

The spread of results in CP1 for the pattern group is 38 to 100, which compares to a spread of 52 to 94 for the control. As might be expected of a group that self-selects on the basis of a feeling that they might require extra assistance, the patterns group represents a greater range, and in particular, a range with a 14% lower minimum value. On the other hand, the spread of results in CP2A shows the opposite tendency, with the greater range, and lower minimum, being achieved by the non-pattern control group, 81% (17 to 98), as against 38% (42 to 80) for the participants. A similar tendency shows up in the average difference between CP1 and CP2A results with the control group achieving, on average, 16.4% less in CP2A, and the patterns group 12.7% less.

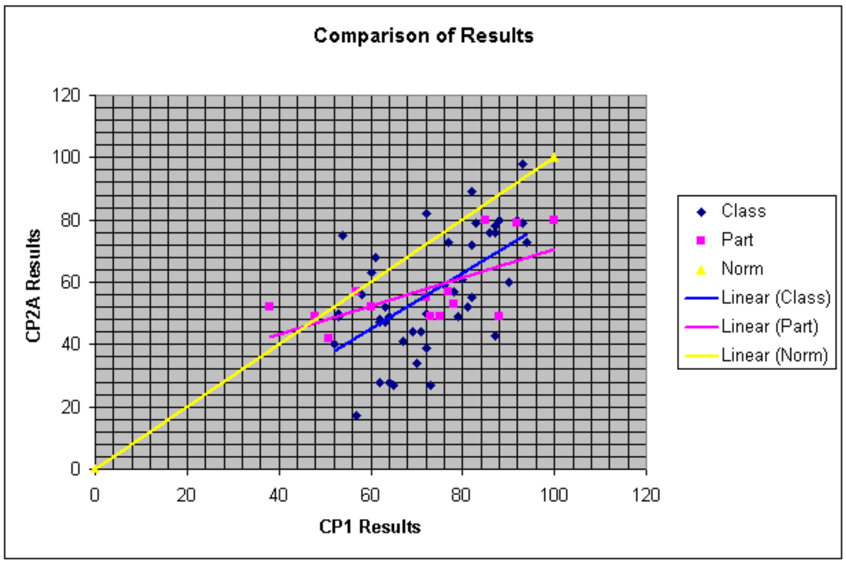


Figure 10.5. Comparison of CP1 and CP2A results

Ideally, instructors would hope to advance the capabilities of their students evenly, such that their results in advanced courses matched the results they obtained in an introductory course. Naturally such an ideal is never actually achieved in practice, nevertheless it provides a benchmark for assessing the progress of students through material of increasing complexity. In the graph shown in Figure 10.5, this benchmark is shown as the diagonal line across the graph from 0, 0 to 100, 100, and represents the 'ideal' correlation for CP2A results, the Y-axis, against CP1 results, the X-axis.

The scores for the students are plotted on the graph, diamonds representing non-pattern control group members and squares the pattern help session participants. Lines of best fit are provided for both groups, that for the control group, 'Class', being roughly parallel to the benchmark but displaced, as would be expected in a real world situation, towards a lower score in the more advanced course, CP2A. The trend for the pattern group, 'Part', is quite different in that, in the lower range, it starts off closer to the benchmark and then trends further away as the scores get higher, ending up below the 'Class' line.

The divergence from the benchmark towards the positive in the lower range is attributable to two participant students who did better in CP2A than CP1.

No students in the lower end of the control group did this, although, further up in the scores, two did manage it. The line for the participants then trends less strongly upwards than the control group suggesting that participation was less effective in maintaining the performance of higher-range students, although this is probably more an artefact of the higher starting point. The important point is that most participants in the middle and high range performed only a little below the trendline for the non-participants, indicating that the difference is marginal. Moreover, the most significant comparison is actual performance compared to the benchmark line, rather than a direct comparison between the two groups. One would expect the usefulness of special instruction to be more apparent in the lower end of the range than the higher because good students require less assistance in general, and this effect would itself be exaggerated by the 'volunteer effect', the nexus between realising that you are struggling and motivation to participate in help sessions.

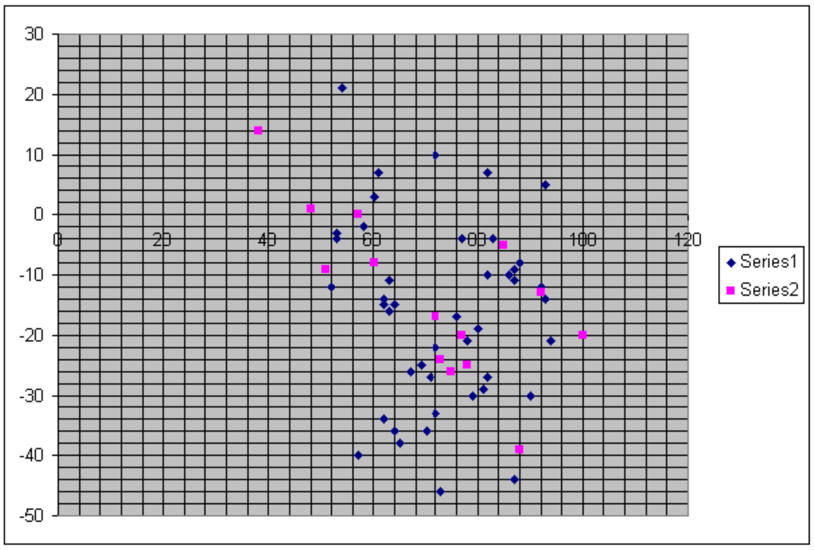


Figure 10.6. Spread of Difference between CP1 and CP2A results

The points on the graph shown in Figure 10.6 represent people who attempted CP2A in 2004 for whom a result in CP1 is available, and who gave permission for their results to be accessed. Not everyone who signed up had completed CP1, presumably having been given status for equivalent work done elsewhere. It is the need for a result in CP1 that explains the fact that all points are within the range 38 - 100% on the X-axis, which represents the CP1 score, the point being that people scoring low marks in CP1 are less likely to attempt CP2A. Two people in this group failed to reach the mark required for an academic supplementary in CP1, and both attended the extra help sessions in CP2A. Both of these people scored a positive difference, indicating that they improved their result in CP2A as compared to CP1.

Both groups, as expected, tended to do worse in CP2A than they did in

CP1, due, most probably to its greater emphasis on demonstrating actual skill at writing programming code as well as the more advanced nature of the material covered. The control group performed more poorly than the pattern group up to approximately the 75% mark, where, as one would expect, the need for, and therefore any beneficial effects of, extra assistance is likely to be marginal. Because the latter group exhibited behaviour that indicated that they were aware, on the basis of their CP1 experience, of a need to seek extra help, some of their better performance is probably attributable to extra motivation - they were motivated to attend extra sessions and therefore, presumably, motivated to work harder at succeeding than the average student. Nevertheless we feel that these results indicate that the pattern material probably did contribute to the participant's tendency to perform, in general, better than their colleges in the control group, particularly in the lower and middle range of programming performance.

All of the figures, the spread of results, the correlations, and the mean differences, seem to suggest that the pattern help sessions did have an effect on performance of programming skill as measured by CP2A assessment. Students in the pattern session group performed closer to the 'ideal' correlation below the credit level, 75%. Moreover, the maximum mean difference for the control group occurs at approximately the 74% CP1 mark, whereas for the pattern group it is approximately 86%, about 12% higher. This suggests that the reductive effect of the more difficult work involved in CP2A cuts in at a higher CP1 mark for participants than non-participants. As with the initial experiment, the number of students involved in the pattern group is probably too small to give definitive results, nevertheless there does seem to have been some beneficial effect, especially for students in the lower and middle range of performance in CP1.

More importantly, the spread of the groups in the difference graph represent a breadth-depth effect - in a given population of students the average impact of the extra difficulty of material and assessment is the same but shifted up towards the higher end of the CP1 result by 12% by the propensity to seek extra assistance. That is, the help session effect seems to be a displacement of the impact of increasing difficulty on result in the second course towards a higher knowledge level, rather than an amelioration of it. Although it is probably still a moot point about the particular nature of the extra assistance, one has to assume that its pattern-based nature does contribute to some degree.

The only way to untangle any effect of the nature of the help from the fact of getting it would be to run the experiment many times with different kinds of help. But one can take this drive for empirical verification too far. No one would seriously suggest that the content of a system of help doesn't matter, only the fact of getting it. This is tantamount to saying that it is impossible to imagine material so confusing as to cause the spread of the assisted group to be shifted to the left rather than the right, and we have all had experiences where some well-meant but unstructured assistance has set one's understanding back, rather than forward. The whole point about the pattern language idea is that a structure, the contextual relationship of language and meaning, is added to an otherwise unstructured, or loosely structured, set of concepts in order to make the system operation clearer and therefore easier. It says of two concepts that here is a relationship between them that is not inherent at the conceptual level but which arises in operation.

In this world some combinations are just more stable, some explanations more coherent, some solutions more elegant, some artistic representations of reality more beautiful. Insisting on an empirical demonstration of these effects misses the point that some 'narratives' simply resonate with the way that the mind works, which is just another way of saying with the way that the world is. Given that the mind is a system in the brain arising from interaction with the world, it should not be surprising that some real-world operations fit its own functional form better than others, and that these 'resonances' are significant in terms of 'understanding'.

Another important aim of this experiment was to test the idea of using a correlation of CP1 and CP2A results as a means of measuring the performance of programming skill. In terms of a methodology to measure the effect of special teaching material on programming performance we feel that these results demonstrate that a correlation of CP1 and CP2A results is quite promising. The results do show a difference in performance between the two groups that, more or less, matches expectation. This method, therefore, clearly provides the potential to maximise the use of volunteers in that a proportion of them is not utilised purely as a control. The main deficiency in the technique employed here is probably the inability to separate the motivation factor from any effect caused by the special material. It would, therefore, be interesting to attempt to remove the self-selecting nature of a volunteer group by presenting the material during one or more of the normal CP2A tutorial or practical streams, and this, in fact, was the basis of the third attempt.

10.5.3 The Third Trial

In moving from a stand alone experiment to providing an extra help component to the second programming course we were attempting to address the attrition issue, the fall off in attendance over time. However, because the extra help sessions are optional, we were again relying on the students' perception of needing extra assistance, and this did not prove to be sufficient motivation. Moreover, it leaves the issue of disentangling improved performance due to motivation and simple attendance from any that might be due to the pattern language exposure. So we felt that the only way to distinguish between the two effects was to attempt to run the experiment utilising a normal component of the course. The only such component that presents such an opportunity is the weekly practical sessions during which students are working on a specified programming task.

As attendance at one of these sessions is a de facto requirement of the course, presenting the attendees at two of the nine sessions run each week with the pattern material and instruction on its use, and basing the assistance given during the session on the pattern process, would be a way of dividing the CP2A class into two groups without having to call for volunteers. This effectively separates out the motivation issue from any potential special material effect and simplifies the recruitment and approval processes - we only needed to get permission of the students for their results to be compared. Comparing the results of the two groups would be based on the same procedure used in the second experiment, the difference between the two attempts being in the nature of the pattern involvement only.

But this difference does mean that the exposure to the pattern material is occurring in an environment where the attention of the students is focussed on what they see as the immediate problem, achieving as many of the checkpoints in the given time as possible. Despite our realisation that this was likely to be a significant problem, we felt that it was important to attempt it on the basis of removing the motivation effect from the results. As it turned out there was another problem, and that was that of the 11 weeks of practicals, five were run as online examinations during which no assistance or interaction was allowed, effectively cutting down the exposure to the extra material, such as it was, to six one hour sessions.

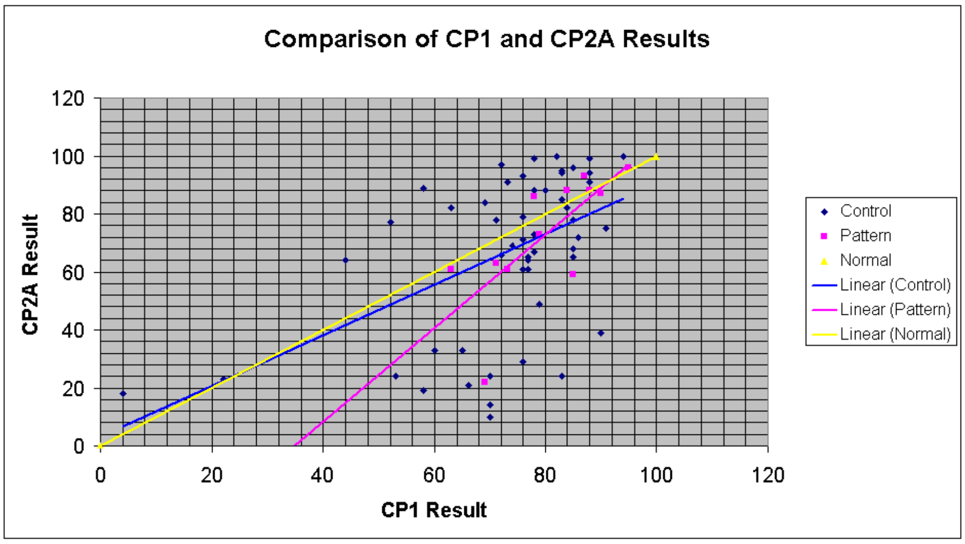


Figure 10.7. Comparison between CP1 and CP2A results

Given these difficulties, it was not surprising that the results, tabulated in Figure 10.7, do not provide any useful information. The spread of results is roughly similar for both the pattern group and the rest of the class indicating that the pattern sessions did not provide the noticeable differences indicated in the first two trials. So the main lesson from the third version seems to be that any attempt to control for factors like motivation would need to be considered carefully. From our point of view this is not too surprising because the pattern language idea is based on a wholistic view of the world, and everything about an individual is a part of the whole of personality. In any case, it is clear that simply utilising a component of the normal stream of instruction for comparative testing of this kind is unlikely to work as the component concerned has its own role to play in instruction, with the attendant pressures and constraints. Adding a further task to the mix, in our case the need to provide instruction in the use of novel material, just overloads the dynamics of the situation. Moreover, instead of reducing the cognitive load as the pattern process should, it is clear that adding the extra material probably increases it, if anything it complicates the "psychological field" for those involved.

10.5.4 The Overall Results

Therefore, the main thing to come out of our attempts is that the standalone experiment is likely to be the easiest way to keep control of enough of the mix of factors to obtain meaningful results. The difficulties encountered in the standalone experiment, our first attempt, were related to recruiting and maintaining a large enough pool of volunteers and providing enough of an exposure to the pattern material to be sure that it was incorporated into the student's thinking about programming. Attempting to address these issues by incorporating the experiment into a stream of help sessions for a second programming course did seem to have advantages in terms of the second issue, but probably exacerbated the attrition rate issue. These two effects are the two sides of the same coin, of course, more exposure requires a greater level of attendance.

We attempted then to split the coin, so to speak, by adding the exposure to an existing component of the second course. This solves the attendance problem as no extra time is involved, but it proved disastrous in terms of exposure. No extra time on top of the normal course commitment means no time for exposure to the special material, it would seem. From this experience it was clear that we took the wrong route in attempting to address the attrition rate issue exposed in the standalone version, but we took the route we did because we felt that the other obvious alternative, paying people for involvement, complicates the motivation issue. In testing the use of material one needs people to be committed to process under test, not simply motivated by payment.

In terms of the standalone version it is unfortunate that we did not think to ask students to produce and hand in a diagram of the pattern sequence they used during the programming test. This would have done two things, it would have made us confident that the material was being used, and it would have given us an indication of what effect it was having on the thinking of the participants. However it would have complicated the comparison of the performance of the two groups, as there is no equivalent representation of the design rather than the coding process that we could have asked of the control group. At the time of conducting the first attempt, however, we had not yet encountered the Concept Map material that demonstrates the usefulness of concept maps (pattern language) diagrams as measures of conceptual understanding, so it may be possible to have the non-special material participants produce a concept map. Of course a concept map is *not* a representation of the pattern sequence derived from the use of a pattern language, nevertheless it may have provided some degree of comparison of the design-level thinking of the two groups.

From our experience, then, the main recommendation we would give anybody attempting to measure the effect of special material on the performance of novice programmers is to conduct the test on the basis of a standalone experiment. There would appear to be only one reasonable way to increase the time committed by the participants without exacerbating the attrition issue, and that is to pay them. This means either that one accepts the complication of motivations involved or attempts some mechanism of recruitment that does not mention payment until *after* the students have volunteered. As the test is mainly about attempting to find a way of helping people learn to program it is fairly important that their involvement is motivated by wanting to learn. Offering payment after the process has begun might be a way of ensuring that they started for that reason rather than for any financial purpose.

However, we also feel that the second attempt taught us an important lesson in terms of comparing programming performance. Because the comparison is made on the results of the students' assessment performance over two 13 week courses, rather than on a single programming test, we feel that it is a more reliable indicator of programming performance. Of course, this is only possible because the two topics involved have assessment regimes that are so clearly based on measuring different factors, the knowledge of the programming language in the first course, and on-line exam-condition programming performance in the second. But, given this setup, the results we obtained showed that any effect due to different pedagogies shows up clearly in the comparison.

Incorporating this method of comparison into a standalone experiment is something that we would have liked to try. Of course it would mean that, as a whole, the experiment would no longer be entirely isolated from the normal teaching regime, but the introduction to any special pedagogical process would be. It might therefore be a way of avoiding the need to utilise some of the precious volunteers as a control, the control being provided by the rest of the class, thus increasing the statistical confidence in any effect detected.

In terms of what we believe about the use of pattern languages in assisting novices to learn how to program we feel that the main thing that our experiments demonstrate is that providing instruction in this form is effective in promoting the *process* of programming rather than simple knowledge of the constructs used. The response of the participants did indicate that they found that the pattern language approach to thinking about a problem was useful in terms of designing a solution. We also noticed that it helped people to express more clearly the aspects of programming that they were finding difficult to understand. So, although our experiments failed to provide sufficient grounds to draw any definite conclusions about the use of pattern languages in programming instruction, we feel that they did show that pattern instruction is useful from the point of view of both the student and the instructor in clarifying what is difficult about coming up with a solution to a programming problem.

Chapter 11

Conclusion

The greatest thing a human soul ever does in this world is to see something and tell what it saw in a plain way. Hundreds of people can talk for one who can think, but thousands can think for one who can see. To see clearly is poetry, prophecy and religion, all in one.

John Ruskin (19th Century Critic)

11.1 The Pedagogical Problem in Programming

There is a fundamental discordance between the idea that understanding derives from the rational discourse with the world that we call reasoning and the means which we use to express it. All symbolic forms of representation are linear in nature, they support the dynamic cut and thrust of discourse hardly at all, and at very best, only indirectly. This is the thrust of Socrates' criticism of anybody "who thinks that he can leave behind him an art in a book, and he who learns it out of a book, and thinks he has got something clear and solid" (Plato 1999). Socrates' point is that understanding is constructed quite differently, it uses an essentially non-linear process " to collect together a multitude of scattered particulars, and, viewing them collectively, bring them all under one single idea, and thereby be enabled to define, and so make it clear what the thing is which is the subject of our inquiry. ... [And] to be able again to subdivide this idea into species, according to nature, and so as not to break any part of it in the cutting, like a bad cook" (Plato translated by John Stuart Mill (Mill 1946)).

In a sense, the formal representation in a symbolic narrative, that is, a linear form, is like Socrates' bad cook, it has 'broken' the essential conceptual structure that is being expressed in any true *understanding* of the world in order to achieve a linear presentation. The linear nature of the expressive form takes precedence over the structure inherent in the content of anything that can truly be called knowledge. This, of course, is a commonplace observation, and limited means of *interrogating* texts, in Socrates' sense of the way that "those who wish to learn"

(Plato translated by John Stuart Mill (Mill 1946) do so, have arisen over time. These are the familiar aids to reading texts such as tables of contents, indices, and the like. The trouble with these is that, like the text itself, they are organised on the basis of an ordering that does not reflect the conceptual order of the material under discussion and so only indirectly enhance the process of understanding. One has to know what it is that one is looking for in order to use these aids effectively, presupposing at least some degree of understanding.

However no such presumption can be made concerning the novice's understanding of programming. The whole project of teaching programming turns on building the conceptual structure that underpins it in the mind of the novice. Unfortunately, the teaching process tends to devolve to the linear form expressed in formal symbolic means. This is, of course, understandable, and indeed, almost unavoidable, as all experience is, by nature, linear in time. Moreover we know, from history, that effective ways of teaching static knowledge can be structured in this way. What history is equally clear on, though, is that effective ways of teaching dynamic skills require developing the sort of knowledge "which is written scientifically in the learners mind" (Mill 1946) that Plato has Socrates talk about.

Understanding derives from the process of "interrogating" the world in order to uncover the hidden connections and dependencies that cause it to be the way that it is. So the scientific and Socratic methods of enquiry are essentially the same, being based on the sequence, *question - hypothesis - test*, with the only real difference being the nature of the testing stage. In the case of the Socratic method one performs thought experiments rather than 'empirical' ones, but the 'thought experiment' is, in one way at least, more precise than its scientific cousin, because negative evidence is not only acceptable, but constitutes a general disproof of the hypothesis and requires its modification. The nature of the periodic 'Kuhnian' revolutions in science corroborate the higher status of the 'thought experiment', even in science, as the data from the paradigm-negating empirical research will be ignored until such time as thinking has "caught up with the data", so to speak. In essence we are involved in a process of interrogating the structure of reality whenever we are *doing anything* that requires understanding.

Given, therefore, the failure of linear means to express conceptual structure, it is not surprising that one of the most common saws of our culture is the one that claims that a picture is worth a thousand words. This is merely saying that most of meaning is structured conceptually and that structure is usually more clearly represented pictorially than narratively. However pictorial form has its own limitations and these revolve around the level of detail that can be pictured. Here, words are essential, but again, adding too many words to a diagram tends to confound, rather than enhance, understanding.

Doing anything that requires understanding is itself a process, and a complicating factor in the programming situation is that the result of the programming process is a program, another process, one that gets executed on a machine. Ultimately, therefore, the process that results in the program has to deal with the execution level details of the machine. But these details mostly have *nothing* to do with the task being automated at the conceptual level. We are, in effect, mixing two domains, the problem and the programming domains, which, at the conceptual level, are related only through the process of *programming* a task in a *problem domain.* This is Dijkstra's point about "the tools we use hav[ing] a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities" (Dijkstra 198). We come to see the problem purely in terms of the tool rather than for what it really is, an element of a different context entirely.

Nevertheless, any program has, in the end, to be executable, so in programming it we have to take account of the machine's execution environment to that extent at least. The wrong turn that we took at the very start of the computer age was to attempt to base our thinking about programming solely on the execution level features, and the history of programming since then has largely been the story of our attempts to move to a mode of thinking that is less driven by execution-level detail. We need a *way of thinking* about problems that exist in the real world that *results in* machine execution, not a way of thinking about problems that exist in the real world *in terms of* machine execution. The sequence of thoughts that we go through in designing a program is *not* the sequence of actions executed by the machine.

There is only one answer to the problem of mixing a domain in which a sequence produced by a process in which understanding is required, the programming domain, with a domain in which the sequence being processed involves exactly *nil* understanding, the machine execution domain, and that is to use a language based on the former rather than the latter, a language based on understanding not blind machine execution. Although computer science has attempted to move away from using the base "machine level instruction set", the "programming languages" that resulted are still tied to "concepts" that can be directed translated into machine level instructions. These "languages" are therefore strictly formal in a way that 'natural' human languages aren't. But, in turn, the push for more "natural language"-like programming languages founders on the requirement that they are machine translatable into executable machine code.

In other words the "machine logic", and therefore any "language" that purports to implement it, is a closed formal system in a way that no "natural language" ever is. The "nature" of a formal system is that it is a completely defined, internally consistent means of invoking logical relationships. It is this confusion between the language used to think about a program, and the 'language' in which it is implemented that bedevils our field.

> The confusion is perhaps most clearly demonstrated by the often expressed opinion that "one cannot use a programming language that has not been implemented". But this is nonsense, of course one can! One can use any well-defined programming language, whether imple-

> mented or not, for writing programs; it is only when you want to use those programs to evoke computations that you need an implementation as well. Being well-defined, rather than being implemented, is a programming language's vital characteristic.
> The above remarks are neither jokes nor puns; on the contrary, they are pertinent to multi-million-dollar mistakes. They imply, for instance, that the development projects - erroneously called "research projects" - aimed at the production of "natural language programming systems" - currently en vogue again - are chasing their own tails.
>
> (Dijkstra 1982, pp. 62-5)

So, at some point, the break with the machine simply *has* to be made. Any language that expresses a solution in conceptual terms has to be based on *understanding* the problem, not executing the solution. What this statement boils down to is that the solution must be expressed in a form that is *not directly translatable into executable machine code*, and up until recently the 'break' has occurred in the mind of the programmer. That is, whatever conceptual form is being used to think about programming exists entirely in the programmer's mind because the only form in which the program is visibly represented is in machine translatable language, the outside observer sees only the *result* of the programming process, not the programming process itself.

What this requires, however, is a thoroughgoing and reliable internalisation of the conceptual structure (understanding) involved, and this is something that only the expert can be expected to have. The basic pedagogical problem in programming, therefore, has always been that we are basing instruction on a "language" that is not *the* "language" used by expert programmers because we know not what that language is, we never see it! Nobody is ever taught the language used in thinking about a programming problem, they develop it through the use of a "language" translatable into executable machine code. No wonder expert programmers are accorded almost guru status, their skill worshipped with almost religious reverence. To the outside observer the source of their skill is entirely invisible, literally 'mystical' in nature.

11.2 The Pattern Language Response

In the end, the reason, the *only* reason, for teaching programming through the use of pattern language is that pattern language underlies programming skill, indeed any human skill. Look at any expert in any field and what you see is a human applying previous experience, which is all that a pattern language is. The roots of the modern mind lie in the transition from an irrational to a rational explanation of the world. Once you start reasoning, that is, seeking the reason for the way that the world is in direct human experience rather than divine purpose, you have no other resources than patterns (of experience) and language (creativity). In that sense, reasoning is built on causal relationships - the ‘cause’ of something is the ‘reason’ it happens.

The whole thrust of the Socratic method is to build understanding by questioning the elements of experience. This is a classic process of evolution. Just as biological form adapts through experience so does mental form. The present is always the creative result derived from the patterns of past experience, there really is *no* other way forward! Life develops not only by learning from its mistakes, but also from its successes, that is, from *all* experience, negative as well as positive.

> Both in science and in biology one expects continuity; small steps are, of course, less dangerous than big leaps. A higher-order mechanism favouring continuity in development will then be an asset. But note that this will be an advantage also for the mechanism. Its own chances of survival increase. Higher order mechanisms are themselves subject to selection in the evolutionary process. Those which lead to rigidity are dangerous and will tend to be eliminated. The same holds for those which trigger off a cascade of untried connections.
> Experience is decisive for the development of epistemic systems at all stages. This holds also for the higher order mechanisms which regulate the limits of experiment. These mechanisms can be regarded as tools of problem solution. Their quality will then be dependent upon earlier attempts at problem solution, and the success of these. The outcome is a technique with empirical foundations.
>
> (Halldén 1997, p. 19)

In specific terms, the pattern language system that we propose addresses the pedagogical problem in programming by generalising the machine-level concepts that are necessary and by including them in a language that contains concepts that are not immediately or directly implementable in machine code. By this means the “programming language” is a *part*, not the *totality*, of the language used for designing the solution to the original problem. It is easy to forget that the original problem specification *always* takes the form, “write a program that does some task on a computer” even if it is not explicitly stated that way. The task of the programmer is therefore to *write the machine specification*, not to *do* the task to be done by the computer.

The point about pattern languages is that they are based on Ausubel’s statement that “the most important single factor influencing learning is what the learner already knows” (Ausubel 1968, p. vi). Pattern languages relate the unknown to what is already known in two ways. Firstly, a pattern is a pattern *because* it is a common experience, and some of the patterns in a language for programming pedagogy will be for everyday concepts like `CHOICE`, `REPETITION`, `SWAP`, and so forth. One very pertinent reason for including such concepts is to build on knowledge that novices will already possess, in programming terms these are metaphors from real life phenomena. But there is an even more important factor that we earlier discussed as the evolution of pattern languages as novices learn.

One of the major shortcomings of current approaches to teaching programming is that concepts are introduced in a linear way which makes little use of the 'natural' connections between them. This then shows up as common mistakes in novice practice such as mixing the process of defining a method with the process of calling it. A pattern language such as Figure 10.3, clearly delineates the difference between them in a way in which the normal presentation has not succeeded in doing. So using an 'evolving' pattern language provides a way of attaching new patterns to the existing structure as they are introduced. This makes the relationships between them clear from day one, they are not just left to appear as though they are isolated ideas plucked out of nowhere for no good reason. The pattern language shows from whence they come in relation to what the novice already knows, and the explanatory material in the pattern itself provides the reason they exist.

There is a very significant juxtaposition here, and it constitutes, we believe, the reason why the pattern language idea resonates in programming in a way in which it resonates in no other field, *including* that in which it arose! And the reason is this. Programming occurs in a meaning-free system, a computer. Because the symbols being manipulated carry meaning for us as humans it is all too easy to overlook the fact that they are entirely meaningless to the logic system. Yet programming is about solving problems that are firmly situated in the real world, the system being automated *reeks* of meaning. Somewhere along the line the meaningless symbols of the logic system have been *given* the *meaning* expressed by the program. And that is *essentially what programming is*, the assignment of meaning, the *giving of context*, to a formulation of essentially meaningless symbols, precisely as the subject of the digit span experiment, discussed in section 9.3, did.

But this is the *ultimate* form of creativity, the creation of meaning. It is not just the manipulation of a few symbols, that's the trivial part. The powerful bit is the *creation* of meaning out of meaninglessness, the symbols are just used to *express* the created meaning *not to create it*! In almost no other field, with the possible exception of mathematics maybe, do the entities being manipulated carry less real world meaning in themselves than programming. But if the power of programming is the *assignment of meaning* then the meaning can only come from one place - the programmer's mind.

But this is an *exceedingly dangerous* proposition, and not only in terms of the programmer's ego either. It makes it sound as though the programmer *creates* the meaning by means of some unique force of personality, the "transition from the problem to the solution is mediated by a conceptual process unique" (Grabow 1983, p. 43) to the individual programmer. So even though the agreed objective of a program is to fulfil the requirements specified for it, "the actual generation of form is presumed discontinuous with its context" (Grabow 1983, p. 43). It's as if the elements (of meaning) that make up the new meaning "created" by the programmer did not exist. This radical discontinuity between the form being generated and its generation is only possible due to the ignoring of context - the solution is presumed to originate *outside* of the context of the problem in the "imagination" of the programmer, as if the imagination of the programmer is not itself a product of context, that is, experience.

This points to the fact that although science deals with phenomena that are objectively given to the mind, it relies on the mind to process the data so given, and it can only deal with the data on the assumption that the phenomena are independent of the mind. If this were not so, that is, if the primary data were taken to be purely a product of the mind, then no reason, science, philosophy or even art would be possible. These activities all depend on meaning, which is a relationship between mind and reality. It makes no sense - there would be no reason - to build a representation of reality, which is what the mind is, if there was no reality to represent. Learning, then, is essentially a dynamic relationship whereby mind is 'fitted' to reality - "a preparation for the pursuits of active life" (Mill 1946, p. 36).

All the 'mental' activities are based on the patterns thrown up by the ever-changing world. What the brain does is to provide the material locus for mental activity - it is "the biology of meaning."

> The biology of meaning includes the entire brain and body, with the history built by experience into bones, muscles, endocrine glands and neural connections. A meaningful state is an activity pattern of the nervous system and body that has a particular focus in the state space of the organism, not in the physical space of the brain. As meaning changes, the focus changes, forming a trajectory that jumps, bobs and weaves like the course of a firefly on a summer night. ... The skills of athletes, dancers and musicians live not only in their synapses, but also in their limbs, fingers and torsos. Neurobiologists who study the molecular basis of learning in synapses tend to overlook the fact that enlarged muscles and honed immune systems have also learned to do their jobs. The strengths of connections between the neurons and the properties of the body are continually shaped by learning and exercise throughout a lifetime. Each of us is born with genetic and cytoplasmic endowments that establish some general limits to the directions and extents of growth in striving for the wholeness of intentionality. A state of meaning then knits together the brain and body in brief time intervals, which, in the language of neurodynamics, form short segments of an itinerant trajectory through the state space of the organism. This state space includes the range of possible actions begun at any moment by the personal history and condition of the organism, its wholeness.

(Freeman 1999, pp. 157-8)

This fundamental connection between mind and reality means that it is not possible, in studying the acquisition of a skill like programming, to avoid the philosophical issues raised by notions like knowledge and creativity, in other words, the role of the mind. If behaviourism has left a legacy to its successor in psychological theory it is a deep scepticism about the need to have any regard for the idea of 'consciousness' and 'mind'. As Daniel Dennett has said, the prevaling attitude during the rebirth of cognitive psychology seems to be a commitment to the irrelevance of 'consciousness' in understanding how the human mind functions.

> Consciousness appears to be the last bastion of occult properties, epiphenomena, immeasurable subjective states - in short, the one area of mind best left to the philosophers who are welcome to it. Let them make fools of themselves trying to corral the 'quicksilver of phenomenology' into respectable theory.
>
> (Dennet 1978, p. 149)

But this, surely, is just the difference between the 'objective' functioning of the brain and the subjective experience of the functioning. In reality it is no different from perception.

> I see light but perceptually know nothing about its nature. I see light as light. But the nature of the mechanisms that subserve my perception of light tell me nothing at all about whether light involves waves or particles, about whether it has a speed, let alone what its speed is, and so on.
>
> (Flanagan 1991, p. 342)

The brain is reacting to the light, and I am experiencing its reaction without knowing anything about how the reaction works at the objective level of brain and light. If there were such a thing as an omniscient neuroscientist observing my brain she would see certain patterns of neural activity and identify them with my experiencing light. But I experience the light in a way that she does not, and the fact that the brain reaction has two aspects that are seemingly incommensurate is not a problem if considered epistemically. There is an analogy here with illusions like the famous face-vase and Necker cube illusions, illustrated in Figure 11.1.

> Strictly speaking, there is just one physical configuration before one's eyes, but it can be seen in either of two ways, as a vase or as a pair of faces in the one case and as a cube with reversed foreground and background in the other case. Gestalt illusions such as these show the possibility that something metaphysically unproblematic (the Necker cube is just a bunch of lines) may be seen, known, or described in two different ways. Gestalt illusions have a further important property: when one is seeing the image in one way, one cannot at the same time see it in the other way. But remember, gestalt illusions are illusions.

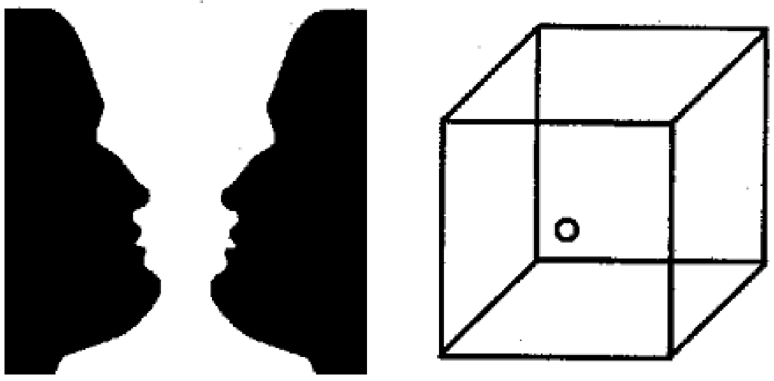


Figure 11.1. Gestalt Illusions

> There is just one thing there. It has two aspects epistemically irreducible to each other, but not irreducible to the whole they comprise, not irreducible in some hoary metaphysical sense.
>
> (Flanagan 1991, pp. 342-3)

So in dealing with a skill, the fact that 'consciousness' is the 'quicksilver of phenomenology' is not a problem. Here we are dealing with the expression of the skill in objective terms, not how it functions at the neuronal level. We need a theory of mind, not a scientific view of the brain. The brain only impacts on our observations of skill performance indirectly, by means of the way that it sets up the mind. In a sense mind functioning is emergent from, not directly related to, brain function. I can safely ignore the subjective 'conscious' process if I can observe the results of performance. A bug in a program is the result of a a mind-level problem, a fault in thinking, not a failure of brain function. The programmer has *mistranslated* some aspect of the real world context of the problem into programming logic because the two are not 'naturally' connected.

But, significantly in regard to this essential absence of real-world context in programming, the pattern language idea is *completely grounded* on the awareness of context. If there is one pivotal idea in pattern language thinking it is that *everything* is fundamentally contextual in nature. In some sense the solution to a problem *exists* already in the context of the problem *because the problem* **is** *the context.* Think of any problem and what you will mostly think of is context. Take, as a cogent example, global warming. Global warming is just a phenomenon, a series of unfolding events. What *makes* it a *problem*, rather than just a phenomenon, is context. The phenomenon is occurring in a context of living systems and it is this fact that gives it its "problem" status. Given no life forms on the planet there would be no *problem*, the increasing temperature of the planet would have no *meaning.* So the *problem* is, in reality, all context - it is the relationships between things that create context, and through context, problems, and indeed, meaning.

What the pattern language idea makes clear is that the meaning expressed by a program written in meaningless symbols comes, *not* from the programmer's head in the sense of a unique individual attribute, but from the representation of the context in which the program is meaningful therein. This is evident even in that most "individual" of human attributes, genius. No genius ever existed in isolation from some set of ideas that, in fact, set the context for the expression of "genius". A great idea is great, not because of the force of individual personality of the person who comes up with it, but because of its effect on our understanding of the world, its *meaning* in terms of the real world - greatness comes from context.

So the juxtaposition between the source of the power of programming, and that of pattern language, is the factor that makes pattern languages so resonant in computer science. The factor that makes a programming logic powerful is its abstractedness, the very fact that the symbols carry meaning *only* in terms of the operation of a machine. This is their only interface with reality. In some sense their power derives from their meaninglessness, their lack of context, in other words, and this is the *exact* and *total* opposite of the power of pattern language. The power of pattern language is that meaning derives from relationships in the real world. It is the abstractedness of a programming system, the very source of its power, that makes the act of programming so difficult. As biological beings we are entities entirely situated in the real world, we are survival, not abstract, systems, meaning is essential to us. So the *only* way that we can deal with an abstract system of logic in terms of generating effects in the real world is through the use of a system that bridges the gap, gives the symbols their meaning, converts them, in fact, from mere marks to symbols.

Because this transformation occurs in the mind, the fact that what is being transformed exists outside of it, is obscured, and it is this obscuring of the actual *situation* of the transformation in the real world that leads to all the dead ends represented by ideas like individual talent and aptitude. Mastery lies in what is being mastered, *not* the master, it is expressed in the artefact through the properties and attributes of the material being shaped into new form. A master in any domain can only do with the material that which the material will allow. Meaning exists "in the the state space of the organism, not in the physical space of the brain" (Freeman 1999, pp. 157). For humans the mind is the substrate in which meaning is expressed but meaning actually *exists* as a relationship in the real world. The relationship between global warning and living biological systems would still exist and have the same essential effect, that is, meaning in biological terms, even if none of the biological systems had minds, that is, were human in that sense. So it is an arrogance on our part to take personal credit for our thinking, we simply provide the means by which the meaning which actually exists as relationships in reality is made abstract and the relationships freed from their concrete form.

Moreover the fact of where the transformation occurs, in the human mind, also obscures its true nature as a language of patterns. The sense of personal achievement, of individual endeavour, hides the fact that new meaning is just a reconfiguration of relationships that already exist. Even the theory of Relativity is just a reformulation of the basic physical relationships between objects that have always been there. Reformulation *requires* a set of data. It is not *abstract*, something that occurs in isolation. The items being shuffled around in the reformulation process are abstract only in form, they would not be symbols if they did not have an essential connection with some factor in the real world. These are *patterns of experience*, *meanings*, not phantoms of random or gratuitous neuronal activity.

It is the lack of a direct connection between the *symbols* and the *meaning* being expressed by the program that is the source of the difficulties in programming. Our belief that pattern language is essential in programming pedagogy derives from the need for a mechanism to bridge the gap, to make the connection between the symbols and the context of the problem, to make them, in fact, *symbolic in the problem domain*, not just phantoms of random electrical activity. What a program has to do is *represent the context* in which the task has meaning in the form of electrical circuits, so programming is, at its most fundamental level, context setting - metaphor.

The clearest exposition of the power of pattern language to give meaning to otherwise meaningless data, to give conceptual context to a random sequence of digits, is the case of SF discussed in Section 9.3, but the 12 year longitudinal study on the use of concept maps (see Section 10.3) backs up their effectiveness in more traditional learning situations. Knowledge is *structured*, not merely random, or linearly sequenced, information. Any process, including the process we call understanding, derives from the *structure* of knowledge, and that is why graphical representations are so ubiquitous in any situation where a process is taking place. Given this fact, the history of the use of such representations in programming, and programming pedagogy is interesting.

11.3 Knowledge and Process

Although, as we have pointed out elsewhere, most changes in programming practice and pedagogy have, by and large, not been driven by empirical research, one area where the results of experimentation does seem to have had some influence on programming practice is the use of flowcharts as an aid to designing programs. At one time flowcharting was ubiquitous, some authorities going so far as to claim that "coding *begins* [emphasis added] with the drawing of a flow diagram" (Goldstein and von Neumann quoted in (Shneiderman) 1977, p. 373). Although flowcharts were seen as a 'high level' definition of the solution to a problem it is not clear that they are anything more than a description of the logic of the solution. That is, they describe a solution that has already been derived and therefore provide little or no guidance in the process of deriving the solution. Like programming code they force the programmer into thinking in strict logic, and while this is feasible for experienced programmers, it is neither desirable or useful, or even possible, for novices.

But the interesting fact about flowcharting is that it has virtually disappeared from programming practice, and this raises the question as to why. Part of the answer lies in the development of 'higher level' languages which attempt to address the 'high level' definition of the solution in terms of a language that is as close as possible to natural forms instead of graphically. Also, of course, the standard flowchart 'language' is based on 'goto' style programming and as structured, and therefore 'goto' free programming, came in, it fell into disuse (Tracy 2003). But this doesn't altogether explain it's disappearance because the flowchart idea can be expressed in forms that support structured style programming. Indeed several versions of such flowchart languages were, in fact, developed, Nassi and Shneiderman (Nassi & Shneiderman 1973) and G. W. Williams (Tracy 2003) promoted examples, but the fact that they didn't catch on suggests that there was more to the demise of the flowchart than simply the rise of structured programming. The use of flowcharts in the early days of programming came, almost certainly, out of circuit design and 'manual' programming of the "plug-panel" kind. The development of the higher-level languages meant that the 'flow' of the program was, more or less, conveyed by the 'linguistic flow' of the program itself, making the use of a flowchart based on any particular flowchart 'language' somewhat redundant.

> We conjecture that detailed flowcharts are merely a redundant presentation of the information contained in the programming language statements. The flowcharts may even be at a disadvantage because they are not as complete (omitting declarations, statement labels, and input/output formats) and require many more pages than do the concise programming language statements.
>
> (Shneiderman et al 1977, p. 380)

Another factor was the almost unconscious realisation that is expressed in the move towards more 'natural' languages, that solving problems requires thinking not just logic, because thinking is mostly language-based.

> Precision is the stock in trade, the *raison d'être* of formal notations. [They] are precise. They tend to be complete. ... [But] what they lack is comprehensibility. ... Almost all formal definitions turn out to embody or describe an implementation of the hardware or software system whose externals they are prescribing.
>
> (Brooks 1983, pp. 63–64)

The development of a solution is informal (heuristic) at least as much as it is formal, and, as an exposition of the flow of the logic, flowcharts provide almost no assistance in terms of *creating* the conceptual solution.

But the fact that attempts to measure the effectiveness of flowcharts in assisting the task of writing programs were, by and large, negative (see (Shneiderman et al 1977)), coupled with the realisation that the effectiveness of graphical representation of algorithms "as an aid to comprehension is questionable" (Waddel &

Cross 1988), probably also played a part in the decline of flowcharting. In a sense, the type of language that supports what we now know as ‘visual programming’ has taken over the claims for the utility of graphical representation of program flow that flowcharts provided, but even here “experiments which compared the comprehensibility of textual with visual programs failed to give support to such claims (Green et al. 1991; Moher et al. 1993). The choice of graphical notations and diagramming techniques is rarely empirically justified”(Chattratichart & Kuljis 2000, p. 46).

The pertinent point about all this is that aids to programming closely based on the implementing “language” seem to be difficult to justify empirically despite the widespread belief held by many experienced programmers that they are useful. It is therefore possible that this paradox is caused, not by the believers being wrong necessarily, but by the difficulty, or even, possibly, the impossibility, of measuring programming performance. Some of the disjunct here lies in the naïve assumption of a strong correspondence between the ability to comprehend program code that is already written, or some aspect of it, such as its logical flow, and the ability to program. But the ability to understand a written representation is *not* the same thing as producing a new one from scratch. Certainly one would expect that the former is necessary for the latter, but one would hardly use the fact that a reader can demonstrate an understanding of published literary works, as a predictor of the ability to produce such a work. There is a correspondence, certainly, but it is not a strong one.

It is possibly not even true that good programmers are necessarily good at understanding written code, these are complimentary but not equivalent skill-sets. Writing programs requires understanding the problem, not other programs. Indeed it may even be the case that understanding programs written by others is *more* difficult than writing a program from a reasonable natural language specification. If the basis of the “software crisis” is the difficulty of meeting the original requirements then this is, at least, a concept-level problem, the failure is a failure to translate natural language into code. The reverse, translating code into understanding has to be a far more unfamiliar procedure, even for experienced programmers. Furthermore it involves the construction of two mental models simultaneously, a model of the program flow and a model of the problem domain (Pennington 1987), usually with little in the way of documentation other than the code itself. At the very least, the programmer starts with some domain level knowledge expressed in the specification. The reverse-engineer has to derive *even that* from the code.

In this respect, patterns can be seen as advantageous in the reverse-engineering process as one of the features of the pattern idea is that it is the ‘conceptual understanding’ behind the code that is being encapsulated, reducing the cognitive effort of building that understanding directly from the code. But there is another factor, and that is that software patterns are the result, in effect, of a process very like the reverse engineering of code - that’s the sense of the “patterns are discovered" notion. In other words the need for novice programmers to digest written programs whole, in order to benefit from the experience of others, is reduced, if not entirely eliminated, by the fact that concepts that are known to have wide applicability will, eventually, find their way into pattern form.

This is probably the main point about patterns, their separation of the concept under discussion from its code implementation. They are means of causing a programmer to think like a problem solver rather than a programmer. It would seem to us that most "aids" to the programmer, like API's and the like, are written in such a manner as to force one to think in programming terms not in terms of the problem one is trying to solve. How can I possibly know, without being told, where to find the method call to convert a String into an int in the Java API? The point is that the API requires preknowledge to make it useful, it forces one to already think like a programmer rather than as a human problem solver trying to deal with an aspect of the real world. To paraphrase John Backus' controversial article, "Can Programming be Liberated from the von Neumann Style?" (Backus 1978), 'to the man with a von Neumann machine everything looks like something that can be dealt with using a programming statement'.

At its most fundamental level the pattern idea encapsulates the fulcrum point of decision making - a pattern reveals that a decision has been made on some point of issue[1]. Once again this is best illustrated by exploring an example and the most revealing, in programming, is the decision about the choice between using a recursive or an iterative (looping) form of repetition. Speak to any experienced programmer about this decision and they will be able to tell you pretty clearly the circumstances in which they would definitely choose recursion and those in which the choice would be to go for a loop. They rarely express any grey areas, even though, when pressed on the point, they admit to being aware that, in most circumstances, the two are, in an objective sense, equivalent in terms of execution and programming convenience. Yet when pressed to explain the reasoning behind their pattern of behaviour, they can't. They know what they do, they are aware at some level that they had a real choice at the point of doing it, yet they cannot explain why they made the choice they did - the decision was made automatically and mostly unconsciously, the sign of a genuine 'ingrained' pattern of behaviour.

What the pattern form tries to do is to give expression to the thinking that lies behind such decision points in programming, to reveal the reasoning that must have occurred at some stage, even though the programmer can no longer give voice to it. Even if it is something that they have picked up in their training, 'inherited' from their instructors so to speak, then this still emphasizes the pattern nature of what is going on. In fact, this is the force behind the idea of using patterns in learning situations. One way or another, students will pick up the patterns of process inherent in their instruction, the thrust of the pedagogical pattern concept is to make the patterns explicit rather than implicit by default. It might

[1]This is why notions of creativity, choice and free will are important in understanding the pattern idea at a theoretical level.

well be that the fact that many experienced programmers cannot explain the reasoning (literally, the reasons for their decision on handling repetition) for or against recursion in particular circumstances is precisely this fact that they have 'inherited' it as a pattern in the most forceful sense of the concept - they simply picked up the patterns of behaviour implied by the 'style' of instruction they received, and their subsequent experience, without even being aware of it.

What this all says is that most, if not all, intellectual activity is post facto justification for what one does. Pressured on the point, programmers will talk about things like whether or not the situation involves a recursively defined concept or not[2], but it is clearly just circular thinking to say that recursive action is driven by recursive definition. Clearly, recursion is one of those fundamental features of thinking, a concept about conceptualization, that is almost impossible to rationalise. Like metaphor we just use it without being aware of it. Who, apart from theoretical mathematicians, for example, is aware that any formal definition of natural numbers is recursive; who apart from theoretical linguists is aware that that the use of language is recursive; and who apart from chaos theorists and fractal mathematicians are aware that aspects of many biological (for example, proteins) and other natural processes involve recursive relations and processes?[3]

Given all this, then, it is probably not surprising that programmers seem to rely more on their 'intuitive' feel for a situation than direct reasoning, in deciding between a recursive or iterative solution. It is not clear that one can ever fully justify it on purely objective grounds, such as execution time or resource usage, for example.

> There are several significant problems with recursion. Mostly it is hard (especially for inexperienced programmers) to think recursively, though many AI specialists claim that in reality recursion is closer to basic human thought processes than other programming methods (such as iteration). There also exists the problem of stack overflow when using some forms of recursion (head recursion.) The other main problem with recursion is that it can be slower to run than simple iteration. Then why use it? It seems that there is always an iterative

[2]Commonly in discussion of recursion one will hear expressions like "inherently recursive concept" (Riley 2002, p. 525), "some problems are easier to solve with recursion ...[because they] involve recursive data structures" (Zimmer 1998), "some procedures are very naturally programmed recursively" (Allen & Dhagat 1999) which amount to little more than the statement "recursion is recursive", and do little to explain when and why recursion is preferable to iteration.

[3]For example a lot of scientific theories model *change* as 'the rate of *change* of the variable is proportional to the variable', a 'definition' that holds as much in Schroedinger's equation as in classical mechanics. Another example is evolution, much of which is modeled by the Fibonacci series, a classic recursive function (a Fibonacci series is one in which each number is the sum of the two numbers that proceed it), so much so, indeed, that it fuels an ongoing dispute between geneticists and mathematicians over primacy. "Geneticists are convinced that the patterns are genetic and the mathematicians keep insisting that they are mathematical" (Stewart 1995, p. 137).

> solution to any problem that can be solved recursively. Is there a difference in computational complexity? No. Is there a difference in the efficiency of execution? Yes, in fact, the recursive version is usually less efficient because of having to push and pop recursions on and off the run-time stack, so iteration is quicker. On the other hand, you might notice that the recursive versions use fewer or no local variables.
> So why use recursion? The answer to our question is predominantly because it is easier to code a recursive solution once one is able to identify that solution. The recursive code is usually smaller, more concise, more elegant, possibly even easier to understand, though that depends on ones thinking style. But also, there are some problems that are very difficult to solve without recursion. Those problems that require backtracking such as searching a maze for a path to an exit or tree based operations ... are best solved recursively. There are also some interesting sorting algorithms that use recursion.
>
> (Danzig n.d.)

Recursion is thus a perfect illustration of the power of a pattern in the decision-making matrix of the design process. But, of course, that which makes it perfect as an illuminator of the power of patterns is the same factor that makes it impossible as an example of a pattern in elucidating pattern form. Because the resulting machine behaviour will be the same in either case, the decision about which of recursion or iteration to use in a particular case comes down to understanding, to the conceptual 'flavour' of the decision point to the particular programmer. The decision is being made on the basis of a purely subjective assessment - does recursive thinking help *me* to *see* the solution to *this* problem? This is probably something that cannot be taught, or even expressed in words, it really is totally a matter of experience.

This does not mean, it should be noted, that the use of recursion cannot be expressed in pattern form. In fact, as Eugene Wallingford has shown with his "Roundabout" Pattern Language (Wallingford 1998), this is entirely possible, and indeed, highly desirable. The fact is that experienced programmers *have* learned to answer the question "does recursive thinking help *me* to *see* the solution to *this* problem?" So it is part of *their* pattern language. But this points to the real force of pattern languages, they are an adjunct to, and facilitator of, experience, not a replacement for it. In the end the use of a pattern is, to some degree, subjective. No pattern language can encapsulate *every* aspect of *every* programmers' experience. All it can do is promote good practice from the feature set of a particular programming system.

But learning to do something via the experience of doing it implies that doing it is feasible for the novice, and it is in this respect that pattern languages are *most* significant. As we have seen, a pattern language encapsulates those elements of a system that keep recurring as the system is used, and it is, insofar as it does that, a facilitator of experience. Therefore, to a significant degree, the patterns in a system are *objective* features of the system, even though they occur in that most subjective of environments, *the programmer's awareness.* The trick is to *make them available to conscious awareness.* For that is the crux of the matter. The phenomena that we classify as intuitive are mostly those where the response *seems* to be driven directly by the sensual input, we are not aware of any reasoning process intervening between the input from the senses and our behaviour.

But, and this is the crux of the matter, it *must* be there. The fact that with increasing experience we become less conscious of the decisions we are making only acts to demonstrate the *absolute* importance of pattern language in human existence. Expertise is the ingraining of the pattern process so deeply in the expression of a skill that it is no longer discernible, even to the expert herself. How else is one to explain the failure of expert programmers in the elucidation of their use of recursion?

Recursion is an objective feature of the programming system, not some phantom that the programmer conjures out of thin air. It is therefore a *pattern of the experience of programming.* All that the pattern language idea says, really, is that making the *patterns of experience* visible is *the only way* that their conscious reuse can be promoted. So the pedagogical pattern idea is just an extension of reuse into instruction. In a sense instruction is nothing more than a form of reuse, but where the reuse is expressed by the novice. What a teacher is saying, even, as we have shown (see Figure 2.1), when patterns are not being explicitly used in instruction, is "this has been found to be a useful feature of programming practice", a *pattern*, "so *you* need to be able to think of it when you are programming." Unfortunately, that's as far as most instructors go. They rely on the practical experience of writing programs to give the novice the awareness of the *when and where* of its application.

Clearly, this is not sufficient, and it is on this point that the current methodology fails. With great persistency, and far too much frustration, a novice *can* learn to program in this way. But it is just not reasonable! It is the most inefficient use of the biological endowment of memory. On its own, biological memory is "episodic" (Tulving & Thompson 1973), the straight *recording* of events in a linear fashion. Therefore the only *reasoning* possible is an almost mechanical *replication* of past experience. Knowledge, if it can even be referred to by that name, is strictly rigid, linear, essentially meaningless in any sense except temporal ordering. It is entirely specific, applicable only in exactly similar circumstances, there is no flavour of general applicability involved. But the real power of reasoning derives from generalised, that is, non-specific, knowledge forms, and these require semantic structure, a web of associations that is independent of the original linear recording of experience.

So all that the pattern language does is to provide the so-called Generic Knowledge Structures (Graesser & Clark 1985, p. 32-3) that illuminate the *reasoning*, to make the connections between the programming concepts explicit so that the *when and where*, as well as the implementation detail, is made obvious. Almost everyone involved in education agrees that "much of what we do as educators is devoted to conveying to the student the cognitive maps that we use for problem solving in a discipline" (Hiltz & Turoff 2005, p. 62), but nobody practices it in any explicit way, because more often that not, the real source of their own facility in programming is a complete mystery to them. It depends on the "automatism" (Shiffrin & Dumais 1981, p. 139) that flows from the generalising of experience, the procedualisation of their knowledge base of which they had almost no conscious awareness (see Chapter 9) as it was an implicit "side effect" of practice.

However, if experts can afford the luxury of not being able to explain the basis of their practice, educators cannot. If the importation of the pattern idea into programming made any sense at all, it did so by pointing out, by default if not explicitly, that even experts need help in using advanced programming concepts. The significance of the GOF book (Gamma et al. 1995) was that although most advanced programmers were probably already writing code equivalent to PROXY and so on, they were virtually "re-inventing the wheel" each time. Putting these concepts in pattern form was thereby a way of making the reuse explicit and therefore easier. If this is true for experts, and patterns were found to be efficacious in this regard, then it must follow that it should also be true for novices with less advanced concepts. There is, perhaps, one proviso, and that is that experts have an established sense of process, they know what they need to do, what they mostly need help with is in remembering the implementation details, *reusing* the concept. So pattern languages, as such, did not arise at this level, both because they are not necessary, and because they are impossible to formulate without the lower level constructs in pattern language form on which to hang them (See Section 7.2).

But the basic thrust of the pattern idea is always pedagogical in nature. Patterns are teaching somebody something, even if it is simply reminding them how a particular concept is implemented. Most of any complex mental task, even in the hands of an expert, is like that. Experience just means that less has to be looked up, not that nothing does. And that is probably the biggest contribution that pattern languages make in the novice programmer context. At least the information that the novice needs is presented in a form that makes finding the appropriate parts of it simple. What the novice needs while designing the solution is a pointer to what is available given the current location (context) in the problem solving process. The pattern language diagram empowers the design process at this contextual level. Having designed the solution, the sequence of patterns that solves the problem at the conceptual level, its implementation is facilitated by the fact that the implementation detail is provided within the pattern form. So the main pattern of behaviour observed in the introductory programming laboratory, the more or less frantic search for *the* example that illustrates what the novice *thinks* is needed to solve the current problem is, at the very least, made easier.

What we are teaching when we are teaching people how to program is a *way of thinking*, a *discipline* in Dijkstra's terms (Dijkstra 1976, p. 60), about a task in terms of automating it. We are giving them, or should be, a language to assist thinking in these terms, "not just a collection of programming constructs" (Sheil 1981, p. 107). Such a language has two requirements, and neither of them suggests that the language be implementable on a computer. The first requirement is that the language be general enough to encompass the wide range of domains in which the tasks automated in the introductory programming environment are likely to come from, and the second is that it assists in generating the understanding required to solve the initial problem. Clearly, the languages currently used in instruction do not fit the bill in this regard. The history of programming is the story of the gradual move away from the logic of the machine, but the situation in the programming classroom will only begin to be eased when the final break is made, when we address the patterns of programming practice, not the logic of execution.

As Alan Kay points out, most of what we call "intelligence" is just a matter of having the appropriate point of view to appreciate the current circumstances for what they are, in other words, *the pattern language for the situation.* In this sense, the human condition is basically the journey involved in learning to be a rational agent, to acquire the skill in identifying the patterns that underly existence, that give the world both meaning and beauty - that is, *form*.

> Watching a famous guy much smarter than I struggle for more than 30 minutes to not quite solve the problem his way (there was a bug) made quite an impression. It brought home to me once again that "point of view is worth 80 IQ points" I wasn't smarter but I had a much better internal thinking tool to amplify my abilities. This incident and others like it made paramount that any tool for children should have great thinking patterns and deep beauty "built-in."
>
> (Kay 1993, p. 70)

The belief that "much of what we do as educators is devoted to conveying to the student the cognitive maps that we use for problem solving in a discipline" (Hiltz & Turoff 2005, p. 62), Kay's "internal thinking tool", is largely misplaced, as is evidenced by the lack of inculcation of such skills in students. We are simply not giving them a better "internal thinking tool" because we expect such a tool to be derived from a simple presentation of the knowledge involved and practice in using it. What is needed is "an explanation of programming skill that integrates ideas about knowledge representations with a strategic model, enabling one to make predictions about how change in knowledge representation might give rise to particular strategies and to the strategy changes associated with developing expertise" (Davies 1994, p. 238), and what we have attempted to do here is to show that *pattern language* is the essential integration of "knowledge representation" (pattern) and a "strategic model" (language).

As we have shown, a pattern language is a representation of the inherent conceptual structure of a field of knowledge, and is thereby an attempt to make visible the "cognitive map" used in solving problems in the domain. Basing instruction on a pattern language provides the means to demonstrate how knowledge is related to practice, it gives the student an indication of the process of solving a problem not just its bare solution as the examples currently used in instruction do. Pattern languages are based on the insight that "designers rarely start from scratch" (Visser quoted in (Glass) 2002) , they base their exploration of the current problem on existing models from prior solutions to similar problems.

So, however it is that the neuronal brain, as an aspect of reality, gives rise to the cognitive maps that we use in our dealings with reality, we do know one thing, and that is, maybe, all that we need to know. Ultimately, that collection of cognitive maps that makes up our model of the world is *created* through experience of the world, there can be no other process at work here, so, at this level, the neuronal details are irrelevant. This is a *map of reality* not a map of neuronal pathways. The true significance of 'mind', indeed the very source of its effective realisation, is that response (behaviour) has been liberated from neuronal dynamics to the extent that response is not based purely on instinct, that is, fixed neural pathways. So the significance of cognitive structure, the "form" of the mental model, is that it is a reflection of the physical, chemical and biological "form" we experience in the world, not just neural structure. Of course, even in its *instinctual* form, behaviour is something that is extremely difficult to relate back to simple configurations of molecules - DNA or neuronal.

> Nobody has the slightest idea of how the mere fact of arranging a few molecules in a particular permutation (a static form) brings about highly integrated *activity*. The problem is far worse here than in morphogenesis, where spatial patterns are the end product. It might be conjectured that the genetic record resembles a sequence of programmed instructions to be 'run' like the punched tape input of a pianola, but this analogy doesn't stand up to close scrutiny. Even instinctual behavioural tasks can be disrupted without catastrophic consequences. An obstacle placed in an ant trail may cause momentary confusion, but the ants soon establish an adjusted strategy to accommodate the new circumstances. ... In other words, the organism cannot be regarded (like the pianola) as a closed system with a completely determined repertoire of activity. An ant must be seen as part of a colony and the colony as part of the environment. The concept of ant behaviour is thus holistic, and only partially dependent on the internal genetic make-up of an individual ant.
>
> (Davies 1989, pp. 187-8)

As the study of metaphor as a cognitive rather than linguistic phenomenon has shown, 'meaning' is the fundamental connection between life and reality - "semantic productivity can be characterized in the same way as morphological productivity, suggesting that form and meaning are organized by the same principles" (Clausner & Croft 1997, p. 247). We are rational productive agents insofar as we have *understanding of the world* (conceptual organisation of meanings). Nothing gets done successfully without it, so everything we do, no matter how abstract it is as a task in itself, has to be based on *understanding of the world*. Ultimately, solutions emerge from the judicious study of discernible reality, because that is where the problem exists - it can only be solved there, in its real context, *not* in the context of a rigidly prescribed programming language in which every possible action is strictly defined.

Conceptual understanding is based on the empirical nature of experience, not on a set of definitions. "Concepts are bundles of statistically reliable features, hence ... having a concept is knowing which properties the things it applies to reliably exhibit (together, perhaps with enough of the structure of the relevant *conceptual hierarchy* [emphasis added] to at least determine how basic the concept is)" (Fodor 1998, p. 92). As we saw in Section 8.3, the trouble with definitions is that "nobody has a bullet-proof definition of, as it might be, 'cow' or 'table' or 'irrigation' or 'pronoun' on offer; not linguists, not philosophers, least of all English-speakers as such" (Fodor 1998, pp. 92-3). Concepts are entirely contextual and context is *entirely a matter of experience*. "Knowledge of reality, whether it is occasioned by perception, language, memory, or anything else, is a result of going beyond the information given. It arises through the interaction of that information with the context in which it is presented, and with the knower's preexisting knowledge" (Ortony 1979, p. 1), that is, the conceptual context of the receiver. So the fabric of concept is the *memory* of repetitive *form* in experience, that is, pattern, and creativity is the combinatory power of a language of patterns in *designing new form*. As Genesis puts it, "language ... is only the beginning of what they will do" (Genesis 11:6).

> The ultimate object of design is form.
> The reason that iron filings placed in a magnetic field exhibit a pattern - or have form, as we say - is that the field they are in is not homogeneous. If the world were totally regular and homogeneous, there would be no forces, and no forms. Everything would be amorphous. But an irregular world tries to compensate for its own irregularities by fitting itself to them, and thereby takes on form. D'Arcy Thompson has even called form the "diagram of forces" for the irregularities. More usually we speak of these irregularities as the functional origins of the form.
> [So our] argument is based on the assumption that physical clarity cannot be achieved in a form until there is first some programmatic clarity in the designer's mind and actions; and that for this to be possible, in turn, the designer must first trace his design problem to its earliest functional origins and be able to find some sort of pattern in them. I shall try to outline a general way of stating design problems which draws attention to these functional origins, and makes their

> pattern reasonably easy to see.
> It is based on the idea that every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble which relates to some particular division of the ensemble into form and context.
>
> (Alexander 1964, pp. 15-6)

Ultimately, as we stated in the Introduction, programming is like living. It is about dealing with the probabilities inherent in a real situation, deciding what you believe to be the case, and as such it is based on hypotheticals, beliefs - partial truths, perhaps - not truth-values as such. As long as we continue to base the teaching of programming on a system based on truth-values we will continue to bypass the fact that the true nature of a problem lies in its context in reality. Once you mix intent (life - in Alexander's sense, as well as the standard meaning) with logic (causation) you have context and pattern, pure logic is no longer a sufficient means on which to create understanding. You need a language of meanings not just the logic of a system of rules as the rules constitute only a part, not the whole, of a living, *creative* system.

In essence, pattern is dynamic form, which explains why it expresses, through language, as a process. But there is a minor difficulty in that it also forms the basis of concept, and in this form there is always a danger of rigidity, of meaning, that is, relationship, becoming frozen - definitional and formal, rather than experiential. Those revolutions in the way that we see the world that need to occur periodically are precisely the result of the need for a pattern of thought that has become frozen into canon to be smashed, so that the essential dynamism of thought, pattern language, can be released anew from established knowledge, to become as fluid, and therefore as creative, as the reality it is meant to represent. All of the static knowledge that we present to novice programmers is as creatively barren as the parchment on which it resides. It can only be transformed into ideal form in the mind, and knowledge in this form is pattern, not formal concept, language-as-understanding, not language-as-communication. Only once it exists in mind/pattern form is creativity, that is, programming, possible. Before they can program we have to bridge this gap in our novices. Programming nous has to become pattern language in their minds, rather than formal canonical knowledge.

But notice the difficulty we have in even talking about this situation. The words in their static form have almost to be *forced* against their nature to express dynamism because much of their informational content, the experiential knowledge they stand for, is formally undefinable. We discussed earlier the basis of this disjunction with the idea of a 'file' as against the dictionary definition of the same idea. It is the difference between concept as a pattern of meaning derived from life, that is, *mythos*, and the restricted form of word as formally defined literal meaning (*logos*). This is translation in the wrong direction. Most of what we call learning is based on extending into more formal means the knowledge we acquire from life. Unfortunately in the systemisation of learning into education, the process, at some point, crosses into ritual, experience loses its dynamism, becomes rote repetition. In that sense we are like Socrates' friend who thinks he has captured art in a book. We act as though we think that programming resides in a programming language formalism.

So pattern language puts back into the mass education system as much of the master-apprentice experience as is likely possible. Of course, a map is *not* the same as having a living exponent of the skill on tap, so to speak, but it has to be an improvement on the disjointed presentation that, necessarily, results from the current pedagogies of skill acquisition, that is, giving students the knowledge in programming language form and expecting the skill in using it to develop through some minimal supervised practise.

11.4 Moral Order

During the development of this thesis we have referred several times to the moral dimension of Alexander's thinking, in particular, his reference to it in his address to the object oriented software community in 1996. Having sidestepped the issue on these occasions it is now important to face up to the implications of his advice because the issue is central to his thinking, not just a peripheral consideration.

> I think that insofar as patterns have become useful tools in the design of software, it helps the task of programming in that way. It is a nice neat format and that is fine. However, that is not all that pattern languages are supposed to do. The pattern language we began creating in the 1970s had other essential features. First, it has a moral component. Second, it has the aim of creating coherence, morphological coherence in the things that are made with it. And third, it is generative: it allows people to create coherence, morally sound objects, and encourages and enables this process because of its emphasis on the created whole.
>
> (Alexander 1999)

Unfortunately, there has been a radical narrowing of meaning in the modern understanding of the word 'moral' that, I believe, has reduced the comprehension and prevented the acceptance of Alexander's ideas more widely. He is, I think, using the idea in the sense of a sort of 'moral order', or as he terms it, 'moral coherence', that flows from the conceptual order made apparent in our dealings with the objective world. When he is talking about the generation of "coherence, morally sound objects," he is addressing the epistemic disjunction, the subjective-objective gap, not the narrow conception of morality as a feature of personal or group belief as it is understood today. As I read it, his thinking goes like this. The natural order that we perceive all around us, as the objective basis of any subjective understanding of the world, provides the *only* means by which we can achieve the wholeness expressed in the ancient ideas of truth, beauty and virtue, the “threefold order of the universe” (Watts 1982, p. 9). All three of these ideas about the world (ideals) express a sort of holistic unity in the way that the subjective awareness in each case matches the objective source of the awareness. Thus truth is the fitting of form to context in terms of apprehending the real facts involved, beauty is the same fitting of form to context in terms of aesthetic value, and virtue the same in the narrow moral sense. But all three together form a whole in that any one of these features is unlikely to occur alone, and that it is in terms of this whole that the phrase ‘moral order’ has currency - that which is most true to its relationship to reality (context) is also both beautiful and virtuous.

It is this same sense of form fitting context that provides the source of Alexander’s feelings about pattern language balancing all the forces involved in a particular scenario, the moral coherence that he talks about, understood in the widest possible sense of ‘moral order’ rather than in the narrow modern sense. It’s about an idea, an artefact of mind process, being true to its roots in the ‘real world’, not just morally correct as in the “right thing to do.” This is why, at a fundamental level, Alexander’s ideas are so difficult to grasp. We no longer live in a Platonic world, as Alexander seems to, where form and context exist as a dynamic living force, the “nature of order” itself. Having separated mind from matter, we are left with no unifying principle, no means to evaluate fit between form and context that is ‘objective’ in spirit, as we are mired in a system where form (the human controllable part of the environment) and context (the part not amenable to direct human control) can only be approached separately. The relativism, even maybe the so-called “clash of civilizations”, of the modern era derives from the radical separation involved. How else is it possible to explain the widespread dominance of beliefs such as ‘beauty lies in the eyes of the beholder’, ‘knowledge is what one believes to be true’ or ‘the end justifies the means’. There is no “moral coherence” here, no resonance of the objective in the subjective, no ‘order of nature’ in the separated aspects as there is in the unified whole.

But, if Alexander, and Plato, are right, then a wide-ranging reintegration of all the aspects of humanity that have become separated over time, mind-body, reason-feeling, logos-mythos, fact-value and so on, is urgently needed because the humanity of a person can only be expressed by the whole, not by the parts separately. Just as a plane can only be a flying machine in terms of the systematic organization of its separate parts, so too can humanity only be expressed holistically. The implication of there existing, in an objective sense, a set of elements (patterns) that are capable of generating all the complex order, not only in nature, but in human affairs as well, is vast, beyond all comprehension almost. Yet we now know that everywhere we look we see this same process at work - pattern language generating form.

The 'moral order' in this sense, then, is nothing more than the reflection of the 'natural order' in human affairs, just as 'conceptual order' is its reflection in the human mind. Just as an individual subjective conceptual structure built on something other than the natural order of things is a delusion, so too is an individual morality so based. Essentially, Alexander's argument about pattern language is that it is *the* means by which 'natural order' is translated into conceptual and moral order rather than the disorder that results from untrammelled subjectivism. There is, indeed, an important sense in which the three levels of order, natural, conceptual and moral are the same - the objective sense - which means that there are two core elements to Alexander's thinking. Firstly he is claiming that conceptual and moral order are generated by a system of rules, and secondly he is claiming that the rules involved in the act of generation are actually *inherent* in the objective world.

> The fact that I'm claiming to put out here is that environments are also generated by systems of rules. They do not have systems of rules which sort of "constrain" their creators. They are actually generated by them. With the onset of computers, for the first time it has actually been possible to study the effect of certain interacting rules. Suppose you take the shape of a wave breaking, for example. You can ask, "Do I understand what is happening?" So you write a set of rules - an algorithm - which is supposed to depict the history of a wave. Then you can run these rules through the computer and generate a pattern of dots on a cathode ray tube. It might be no more than a dozen rules, but if you keep going through those rules, over and over again, in different combinations of sequences, and you are successful, you will actually see this pattern of dots forming a breaking wave. Now when we talk about things like the breaking of a wave we might be up to a dozen rules. In the case of an organism, there are about fifty thousand genes responsible for an incredible number of interactive rules. In the case of environments, there are hundreds. This kind of complexity cannot be accounted for by the kind of mathematics [normally considered]. And indeed, it is only by studying the process which consists of the interaction of the set of rules that you can begin to generate that kind of complexity. So the fact that the environment is created like this - generated like this - is a very remarkable thing. It is miraculous and beautiful.
>
> (Alexander quoted in (Grabow) 1983, pp. 48-9)

The important thing here is that the conceptual structure - the computer simulation in this case - is built on the mathematical relations that exist in the objective world, and that it is these relationships that generate the form, not any individual or subjective factor. But even more startling is the implications of this in terms of the expression of the subjective.

> Now, once we get into linguistic systems, and pattern languages specifically, you not only have these very complex rules that generate things but you also have the power of choice - so that you are free to make something that has not been made before by allowing the system of rules in your mind to do it. This is another step which goes further than saying that, indeed, nature is produced by interacting rules. In a linguistic system or in a pattern language you not only have very complex sets of interacting rules but you have choice. You can say any sentence you want to say at a particular moment in order to make a response to something and, similarly, you can create something that is appropriate to a particular environmental situation which was never made before. But it is the structure of your rule system or language that is enabling you to do this. And that same structure ultimately resides in the finished product, although you have still made it and have created a thing never before created in that specific framework. But to realize that there is no opposition between the immense creative power and the power of the rules - that is difficult to grasp.
>
> (Alexander quoted in (Grabow) 1983, p. 49)

It is difficult to grasp, indeed! These two sources of power are the basis of the long-running free-will - determinism debate, after all. But it follows from the proposition that conceptual and moral order flows from ‘natural order’ that the expression of free will, choice, is only *rational* insofar as it reflects objective reality. I can choose to believe that I have some sort of supernatural power of flight but if I try to express it by jumping from a 300 metre cliff, I will die. The point here that the choices we make are *generated* by the objective status of our existence, not just our subjective awareness. We are, after all, organisms that were generated by nature, is it really so surprising that our consciousness is driven by our relationship with reality? However, unlike the computer simulation of the wave, the generative forces involved in linguistics and pattern languages are not just *reflections* of rules that exist in reality they are *inherent* in reality.

> In the case of linguistics and genetics, we are saying that the rules actually exist. They are not just a conceptual model to explain what is going on - they are in the real thing, although you have to discover them by inference. This is very important in the case of the environment because what I am claiming to have discovered is that there are rules operating in this same way in the environment. I am not saying that this is a handy simulation. I am saying that these rules are actually there, in people’s heads, and are responsible for the way the environment gets its structure. ... Pattern languages are not like [Chomsky’s] generative grammars. What they are like is the semantic structure, the really interesting part of language and which only a few people have begun to study. The structure which connects words together - such as ‘fire” being connected to “burn,” “red,” and

> "passion" - is much more like the structure which connects patterns together in a pattern language. So pattern languages are not so much analogous to generative grammars as they are to the real heart structure of language which has hardly been described yet.
>
> (Alexander quoted in (Grabow) 1983, p. 50)

It is the claim that it is the "real heart structure of language", the semantic connections that exist between concepts and which can only derive from experience of the 'natural order' of the universe[4], that powers all creativity, comes as a total shock to those of us who live in the modern era. It goes against everything we believe about ourselves and our place in the overall scheme of things.

> The idea that the structure comes from these languages rather than from the creative brilliance of designers is initially repulsive. Architects imagine they are creating buildings and, by extension, towns or parts of towns and that these entities are the products of the fertility of the imagination. To have a theory which claims that there are these systems of rules and that we, by embodying these rules, produce particular versions of the structure implicit in the rules - but no more than versions - and that it is really the implicit structure which governs, is pretty much of a shock to the ego. Even lay people tend to think that architects control the environment. The basic attitude is that architects bring order into an otherwise chaotic situation - instead of recognizing that the order comes through this system of rules which, in some version or other, exists anyway. It's the same difficulty one has in understanding that a bird can be made from a set of rules. People just won't believe it.
>
> (Alexander quoted in (Grabow) 1983, p. 46)

It is this element of the unbelievable, the *mystical* quality of the claims, that expresses the sense of 'moral order' that Alexander, I believe, is using when he talks about pattern language generating "coherence, morally sound objects."

Because we experience conceptual, aesthetic and moral factors primarily as feelings - truth discovery as excitement and elegance, aesthetic appreciation as pleasure and the numinous, and virtue as righteousness and guilt or shame, for example - we tend to overlook the objective factors that give rise to the feelings. In the type of thinking that we call 'cognition', the connection between the real world entity and the expression of it in our mind is centre stage, yet even so we find it difficult to maintain a realistic relation between the two. How much more difficult is it then to relate a *feeling* back to its causal roots in reality when the fit between form and context to which we are reacting emotionally is not as readily available to normal cognitive awareness? Yet the two apparently different processes, the emotional reaction and the creative imagination, are aspects of

[4]'Fire' 'burns' and is 'red' in colour. 'Passion' feels, in some fashion, hot like 'fire', that is, it seems to 'burn.' These are simple facts of everyday experience.

the same system, mind, a system, moreover, that only makes sense insofar as it informs us about some objective feature of our circumstance, *illuminates context* in fact. Given this essential congruity in space, time and purpose, it is difficult to accept that these features of thinking can be fundamentally different in spirit. And it is this, I believe, that is the crux of Alexander's point about 'moral order'.

So Alexander's claim for moral coherence is not a simple call for what pattern practitioners do to be referred back to some narrow measure of what is 'morally correct', it is central to how a pattern language actually works to generate the Quality Without A Name, to enable people to live their lives to the full. It is a part of the package not an external measuring stick or an optional add-on. Of course, it could be argued that in a predominately technical field, such as programming, these non-technical considerations have no place, but this is precisely the point that Alexander makes in respect of his own field, that it is the total concentration on the technical architectural details that has led to the contemporary blind alley in building design - the excessive consideration given to the technical order of the field results in 'moral disorder' instead of 'moral order' in terms of providing an environment for community life. Here, we are simply making the same point in regard to programming. The total reliance on the logical structure imposed by the nature of the computer produces a similar result in programming and it is not surprising that the 'moral disorder' manifests primarily as a misfit between the way that the mind works and the programming task just as, in architecture, the misfit presents at the interface between community life and the architectural superstructure that is *meant* to support it. In programming it is only the fact that the experience gained over time disguises the misfit that the field has advanced as much as it has. Moreover, it might well be that much of the continuous atmosphere of crisis in software engineering over the last few decades is also a manifestation of the basic misalignment between mind and strict logic, and it is only because of the modern narrow conception of what is 'moral' that the connection with Alexander's ideas concerning 'moral order' has been missed.

Any "problem in a context" is an indication of conceptual, and therefore, in Alexander's sense, moral confusion. Something about the human relationship with the world, mind, is not fitting its context, not meshing well with that part of the human environment that is not directly amenable to human action. The answer is new form, form better fitted to the problematic context. In our field the problem we face is the massive difficulty that most people have in learning to program, and the context in this case, the part of the environment over which we have little or no control, is the fundamental logical and 'mechanical' nature of the machine (the technical aspect). The only way to approach this problem is to construct new form, that is, to better fit the mind to the context. Mostly, in computer science, we have expended the effort of the last 50 years in a wasteful, and ultimately doomed, attempt to change the context, that is to modify the nature of the machine, at least in how it presents to the programmer, leading to the sort of blind alleys, the "multi-million-dollar mistakes" pointed to by Dijkstra (Dijkstra 1982, pp. 62-5). Admittedly, some effort has been devoted to reconstructing the form, how the mind approaches the machine, but, as it has not been based on pattern language it too has failed to address the heart of the matter - it has not tapped the generative power of objective order.

When Alexander exhorted us at OOPSLA in 1996 to take the moral dimension of his work seriously, he could not, and indeed he said as much, be aware of the particular manifestation of bad fit and moral confusion in our domain. Yet, it is, in fact, exactly the same problem in both fields, his and ours - a basic misalignment between human organism and mechanical order that results in 'moral' disorder. In the clamour for technical progress, any sense of moral purpose, in both meanings of that word, has been lost. And as he points out, without the language half of his conception, the web of semantic connections between the patterns, there can be no generation of "coherence," no "morally sound" (whole) objects created. We are left, therefore, to depend, as we always have depended in Computer Science, on the 'native' creativity of the individual programmer, a dependence that the experience of 5 decades in the field with novices demonstrates is entirely unsatisfactory.

11.5 Epilogue

The essence of our argument is that there is a fundamental correspondence between programming as the design of an artefact for human purpose and biology as the design of an organism for survival. Programming-as-design, like evolution-as-design, is an *informal* process that happens to be implemented in a formal symbolic system. In other words, evolution proceeds on the basis of patterns of life experience, fitness, adaptation, and so on, *NOT* the formal *programming language* of the genetic system that underlies the implementation of the 'design' exposed by the experience of life, the pattern process. In the end, meaning can only be expressed at the informal level of design, *not* the actual coding of implementation. The problem in programming, like that in life, is the fundamental problem of 'meaning', the basic 'epistemic cut' between reality and its representation in a system that is attempting to 'understand' reality. It is pattern that forms the language of meaning, not DNA molecules or electronic switches, which is why this is a philosophical issue not one of communication. If the basis of metaphysics is the question "why is the world the way it appears to be?", then the particular metaphysical problem we face is "why is programming so difficult to learn?" Learning is the creation of meaning. It is the learner who crosses the subjective-objective gap, who 'understands' (creates meaning). And if we might be allowed to paraphrase Douglas Hofstadter, understanding is not a personal issue, it is a matter of having the proper representation of concepts in a mind(Hofstadter 1985, p. 528), in other words, a pattern language.

The force of the pattern language idea is this recognition of the fundamental process, the fitting of form to context, *meaning*, through design for function or purpose, and there are many other aspects of these issues that we would have liked to pursue. Like, for example, the relationship between pattern language and the almost mystical feel for motivating learning exhibited by those really great teachers that we have all encountered on our way through life. As with the aesthetic and moral dimensions of the pattern language idea that Alexander discusses, and which we treat only in passing, we feel that greatness in teaching, indeed maybe even greatness in general terms, is an aspect of Alexander's "Quality Without a Name" that derives from an intuitive grasp of the pattern process, where the intuition demonstrated is a product of experience, not an 'innate' characteristic.

Appendix A

Example Patterns

This appendix contains 4 patterns that appear in the pattern languages used in the experiment detailed in Chapter 10 and in the step through example in Chapter 7. The first two, `Ojects Everywhere` and `Director`, are in the pattern language for identifying the objects needed for a program (see Figure 6.4), and `Class-as-Blueprint` and `Action Method` are part of the pattern language for developing a class (see Figure 6.6). These patterns were adapted from various sources, including David Riley, “The Object of Java” and the Pedagogical Pattern site of Joe Bergin and Eugene Wallingford, for use in the first programming course at Flinders University.

`Objects Everywhere`

Problem
You need to program a computer so that it performs a particular task. The task comprises many possible scenarios, and the detailed behaviour depends on the particular circumstances in a complex way.
Solution
Think of using the computer to construct a model of the way the task might be performed as a real-world activity. Decompose the task so that you can think of it as being carried out by separate interacting objects. Think of each of the objects (identified by nouns in the task specification) as having a set of attributes (properties) that represent its state, and a set of actions (behaviours) that define the way it interacts with other objects. Every object that exists belongs to one or more categories, that is, groups of objects with the same properties and behaviours. Thus a particular car is just one example (instance) of a category called car. Java handles the concept of categories by means of the class mechanism, that is, any object in java is an instance of a class just as any object in the real world is an instance of a category. See *Class*.

Related patterns
Clients & Servers suggests a way of thinking that helps to identify potential objects. *Director* recommends creating a single object that has overall responsibility for other objects. *Action Object* shows how to create objects that respond to user input. *Class* discusses the way that categories of objects are represented in Java. *Class-as-Blueprint* shows how to define the behaviour of the objects.

Director

Problem
You are writing a program that will need lots of objects, all interacting with each other. You need a way to get the program started and maintain overall control.
Solution
Design an object that will act as a director, creating other needed objects and coordinating their interaction. Then you can start the program simply by creating a single instance of the director. The pattern is named after the director of a stage play or film, who is responsible for choosing and hiring actors, telling them what to do, and maintaining overall responsibility for what goes on.
Related patterns
Clients & Servers suggests a way of identifying and refining possible objects and their roles. *Director-as-Handler* shows how to design a director to respond to external events.

Class-as-Blueprint

Problem
You need to define the characteristics of the objects that will be used in running a program. The program will need lots of different kinds of objects, and lots of instances of each kind.
Solution
Organise the objects that will be needed for the program into groups of like kinds, where objects in each group have the same attributes and behaviours. For each group, define a class that will act as a blueprint from which you can create the object instances. The class defines the attributes and methods that each object of the class will have, and it provides constructors that tell how to construct new instances of the class.
Related patterns

Attribute Variable shows how to define object attributes, and *Method* shows how to define object actions. *Instance Constructor* shows how to specify actions that should occur then instances are created.

Method

Problem
You're defining a class for constructing objects that have actions. Each object created from the class will be able to perform the same set of actions, which define the services that the objects provide to their clients. Sometimes there are different ways to request an action, each distinguished by different action data.
Solution
Define each action using a method declaration. The method specifies both the name for the action and the sequence of instructions to carry it out. If there are multiple ways to request the action, use multiple method declarations with the same name, each distinguished by its parameter signature. Using a method declaration to define an action means that the algorithm (strategy) used by each instance will be the same because the instruction sequence is the same for each. However, the consequences of different instances performing the method will typically be different because each object typically has different attribute values, and the effect of a particular instruction (or indeed which instruction to execute next) often depends on the values of attributes.
Code Example

Problem:
You are writing a program to administer a printer which has several options.
Solution:
Each option will require a printing job to be handled differently so there will be a separate method with different signatures for each option.

```
public class Printer {
          . . .
   public void print() {//code for plain printing}
   public void print(String paperSize) {//code for paper size}
   public void print(int copies) {//code for multiple copies}
          . . .
}
```

Every instance of class Printer will have copies of these methods. Calling a method requires using its exact signature, i.e passing it the correct type of parameter as specified in the parameter list. So, for example, the methods defined above would be called as follows:

```
Printer max = new Printer();  //construct an instance
max.print();       //call the method for plain printing
max.print(A4);   //call the method for printing on A4paper
max.print(7);      //call the method for printing 7 copies
```

Note how the actual value passed as a parameter in the method call matches the type specified in the parameter list. Thus the first method call has no parameters (), the second is passed a String value (A4), and the third an int (7). The method call is always related to the signature of its definition in this way. Sometimes you are required to work in reverse that is, you are given a method call, say:

```
max.print(A3, 4, Color.blue);
```

From this code you can tell that you need to write a method called print that takes three parameters, the first being a String, the second an int, and the third a Color object. This means that you can write the method specification using the following signature:

```
public void print(String paperSize, int copies, Color c) {
   //code for printing multiple copies to a specified
   //paper size and colour
}
```

Related patterns

Action Sequence shows how behaviour is composed from primitive instructions. *Elements of Style* discuses the issue of distinguishing overloaded names (such as multiple methods of the same name) based on parameter signatures.

Bibliography

ACM (2004), ‘Technews’.
URL: *http://www.acm.org/technews/articles/2004-6/1129m.html#item2*

Adelson, B., Littman, D., Ehrlich, K., Black, J. & Soloway, E. (1984), Novice-expert differences in software design., *in* B. Shakel, ed., ‘Interact ’84 First IFIP Conference on Human-Computer Interaction’, Elsevier, North-Holland, pp. 473–478.

Alexander, C. (1964), *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, Massachusetts.

Alexander, C. (1977), *A Pattern Language*, first edn, Oxford University Press, New York.

Alexander, C. (1979), *The Timeless Way of Building*, first edn, Oxford University Press, New York.

Alexander, C. (1999), ‘The Origins of Pattern Theory’, *IEEE Software* **16**(5), 71 – 82.

Alexander, C. (2002*a*), *Nature of Order*, Vol. 2, pre-press edn, Oxford University Press, New York.

Alexander, C. (2002*b*), *Nature of Order*, Vol. 1, pre-press edn, Oxford University Press, New York.

Alexander, C., Silverstein, M., Angel, S., Ishikawa, S. & Abrams, D. (1975), *The Oregon Experiment*, first edn, Oxford University Press, New York.

Allen, C. & Dhagat, M. (1999), ‘When to use Recursion/When to use Iteration’, At http://grimpeur.tamu.edu/ colin/lp/node37.html.

Anderson, J. R. (1981), *Cognitive Skills and their Acquisition*, L. Erlbaum Associates, Hillsdale, N.J.

Anderson, J. R., Greeno, J. G., Kline, P. J. & Neves, D. M. (1981), Acquisition of Problem Solving Skill, *in* J. R. Anderson, ed., ‘Cognitive Skills and their Acquisition’, Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 191–230.

Appleton, B. (1998), 'Brad appleton's software patterns links'.
URL: *www.cmcrossroads.com/bradapp/links/sw-pats.html*

Appleton, B. (1999), 'Patterns in a nutshell: The "bare essentials" of software patterns'.
URL: *www.enteract.com/ bradapp/docs/patterns-nutshell.html*

Appleton, B. (2000*a*), 'On the nature of 'The Nature of Order"'.
URL: *www.enteract.com/ bradapp/docs/NoNoO.html*

Appleton, B. (2000*b*), 'Patterns and software: Essential concepts and terminology'.
URL: *www.enteract.com/ bradapp/docs/patterns-intro.html*

Arizona State University (2000), 'Lecture calls for a makeover for maths'.
URL: *http://ether.asu.edu/peekaboo/*

Association GREX (n.d.), 'Thinking and the new psychology: Imageless thought'.
URL: *www.es-conseil.fr/GREX/textes%20introspection/mandler %20chapitre%204.pdf*

Astrachan, O. (n.d.), 'Elementary Patterns: ChiliPlop 98 Hot Topic Proposal'.
URL: *www.cs.duke.edu/ ola/chiliplop/*

Atkinson, R. K., Derry, S. J., Renkl, A. & Wortham, D. (2000), 'Learning from examples: instructional principles from the worked examples research', *Review of Educational Research* **70**, 181–214.

Attneave, F. (1959), *Applications of Information Theory to Psychology*, Holt, Rinehart and Winston, New York.

Ausubel, D. P. (1968), *Educational Psychology: A Cognitive View*, 1st edn, Holt, Rinehart and Winston, New York.

Ausubel, D. P., Novak, J. D. & Hanesian, H. (1978), *Educational Psychology: A Cognitive View*, 2nd edn, Holt, Rinehart and Winston, New York.

Ayen, W. E. & Grier, S. (1983), A new environment for teaching introductory computer science, *in* 'Proceedings of the 14th SIGCSE technical symposium on Computer Science Education', pp. 258–264.

Bachelder, B. L. & Denny, M. R. (1977), 'A theory of intelligence: II. the role of span in a variety of intellectual tasks', *Intelligence* **1**, 237–256.

Backus, J. (1978), 'Can programming be liberated from the von neumann style?', *Communications of the ACM* **21**(8), 613–641.

Bacon, F. (1996), 'The great instauration'.
URL: *http://history.hanover.edu/courses/excerpts/111bac.html*

Bal, H. E. & Grune, D. (1994), *Programming Language Essentials*, Addison-Wesley, Wokingham, England.

Baron, N. S. (1986), *Computer Languages: A guide for the Perplexed*, Penguin Books, London.

Basque, J., Pudelko, B. & Léonard, M. (2004), 'Collaborative knowledge modeling between experts and novices: A strategy to support the transfer of expertise in an organization'.
URL: *http://www.cmc.ihmc.us/papers/cmc2004-215.pdf*

Bateson, G. (1973), *Steps to an Ecology of Mind*, Granada Publishing, London.

Bateson, G. (1979), *Mind and Nature: A Necessary Unity*, Wildwood House, London.

Beatty, J. (1995), *Principles of Behavioral Neuroscience*, Brown and Benchmark, Madison.

Bergin, J. (2000), 'Why procedural is the wrong first paradigm if oop is the goal'.
URL: *http://csis.pace.edu/∼bergin/papers/Whynotproceduralfirst.html*

Bergin, J. (2002), 'Coding at the lowest level, coding patterns for java beginners version 6'. There are many links to other relevant material on Joe's home page: http://csis.pace.edu/∼bergin/.
URL: *http://csis.pace.edu/∼bergin/ patterns/codingpatterns.html*

Bergin, J. (2005), 'Joseph Bergin home page'.
URL: *http://csis.pace.edu/∼bergin/*

Bergin, J. (n.d.), 'Simple design patterns'. There are many links to other relevant material on Joe's home page: http://csis.pace.edu/∼bergin/.
URL: *http://csis.pace.edu/∼bergin/papers/SimpleDesignPatterns.html*

Bergin, J., Stehlik, M., Roberts, J. & Pattis, R. (1997), *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*, John Wiley & Sons, New York.

Bergin, J. et al. (1999), Resources for next generation introductory CS courses: report of the ITiCSE'99 working group on resources for the next generation CS 1 course, *in* 'Working group reports from ITiCSE on Innovation and technology in computer science education', ACM Press, pp. 101–105.

Bertholf, C. F. & Scholtz, J. (1993), Program comprehension of literate programs by novice programmers, *in* C. R. Cook, J. C. Scholtz & J. C. Spohrer, eds, 'Empirical Studies of Programmers: Fifth Workshop', Ablex, Norwood, N.J.

Biddle, R. & Mercer, R. (1997), Resources for early object design education, *in* 'Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (Addendum)', ACM Press, pp. 99–104.

Bishop-Clark, C. (1995), 'Cognitive style, personality, and computer programming', *Computers in Human Behavior* **11**, 241–260.

Blacker, D. (1994), 'Education as the normative dimension of philosophical hermeneutics'.
URL: *http://www.ed.uiuc.edu/EPS/PES-Yearbook/93_docs/BLACKER.HTM*

Blackmore, S. (1994), 'Alien abduction', *New Scientist* **144**(1952), 29–31.

Blackwell, A. F. (2002), What is programming?, *in* J. Kuljis, L. Baldwin & R. Scoble, eds, '14th Annual Workshop of the Psychology of Programming Interest Group', Psychology of Programming Interest Group.
URL: *http://www.ppig.org/workshops/14th-programme.html*

Blackwell, A., Robinson, P., Roast, C. & Green, T. (2002), Cognitive models of programming-like activity, *in* 'CHI '02: CHI '02 extended abstracts on Human factors in computing systems', ACM Press, New York, NY, USA, pp. 910–911.

Blaschke, R. (2000), 'Learning a new programming language', At http://www.rblasch.org/studies/cs665/essay/essay.rtf.

Bloom, B. S. (1956), *Taxonomy of Educational Objectives : The Classification of Educational Goals : Handbook 1 : Cognitive domain*, David McKay Co., New York.

Bloom, B. S. (1971), *Taxonomy of Educational Objectives : The Classification of Educational Goals : Handbook 1 : Cognitive domain*, David McKay Co., New York.

Blum, B. I. (1996), *Beyond Programming: Toward a New Era of Design*, Oxford University Press, New York.

Boas, F. (1963), *The Mind of Primitive Man*, Free Press, New York.

Bohm, D. (1962), *Quanta and Reality: A Symposium*, Hutchinson & Co Ltd, London.

Bohm, D. (1980), *Wholeness and the Implicate Order*, Routledge & Kegan Paul Ltd., London.

Booch, G. (1999), 'On patterns: straight talk on what makes patterns work'.
URL: *http://web2.infotrac.galegroup.com*

Bower, B. (1997), ‘Humanity’s imprecision vision’, *Science News* **152**(26).

Boyle, D. G. (1971), *Language and Thinking in Human Development*, Hutchinson Co Ltd, London.

Brady, J. M. (1977), *The Theory of Computer Science: A Programming Approach*, Chapman and Hall, London.

Bransford, J. D., Brown, A. L., & Cocking, R. R. (2000), *How People Learn: Brain, Mind, Experience, and School*, National Academy Press, Washington, D.C.

Brinck, I. (1997), The gist of creativity, *in* A. E. Andersson & N.-E. Sahlin, eds, ‘The Complexity of Creativity’, Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 5–16.

Bringuier, J. (1980), *Conversations with Jean Piaget*, University of Chicago Press, Chicago.

Britton, J. (1970), *Language and Learning*, Penguin Books Ltd, Harmondsworth, Middlesex.

Brooks, F. P. (1982), Ten pounds in a five-pound sack, *in* ‘The Mythical Man-Month: Essays on Software Engineering’, Addison-Wesley, Reading, Massachusett.

Brooks, R. (1978), Using a behavioral theory of program comprehension in software engineering, *in* ‘Proceedings 3rd Int. Conference on Software Engineering’, IEEE, New York, pp. 196–201.

Brooks, R. (1983), ‘Towards a theory of the comprehension of computer programs’, *International Journal of Man-Machine Studies* **18**, 543–554.

Calder, N. (1979), *Einstein’s Universe: A Guide to the Theory of Relativity*, Penguin Books Ltd., Harmondsworth, Middlesex.

Carroll, J. (1995), *Evolution and Literary Theory*, University of Missouri Press, Columbia, Missouri.

Carroll, J. M. (1996), ‘Ten misconceptions about minimalism’, *IEE Transactions on Professional Communication* **39**(2).

Chabria, A. (2005), ‘Musings from a mouse’.
URL: *http://www.technologyreview.com/articles/05/08/wo/wo_081505-chabria.0.asp*

Chase, W. G. & Chi, M. T. H. (1980), Cognitive skill: Implications for spatial skill in large-scale environments, *in* J. H. Harvey, ed., ‘Cognition, social behavior, and the environment’, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Chase, W. G. & Ericsson, K. A. (1981), Skilled Memory, *in* J. R. Anderson, ed., ‘Cognitive Skills and their Acquisition’, Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 141–189.

Chase, W. G. & Simon, H. A. (1973), ‘Perception in chess’, *Cognitive Psychology* **4**, 55–81.

Chattratichart, J. & Kuljis, J. (2000), An assessment of visual representations for the flow of control, *in* ‘Proceedings of the 12th annual meeting of the Psychology in Programming Group’, pp. 45–60.
URL: *www.ppig.org/papers/12th-chattratichart.pdf*

Chi, M. T. H., Feltovich, P. J. & Glaser, R. (1981), ‘Categorisation and representation of physics problems by experts and novices’, *Cognitive Science* **5**, 121–152.

Churchland, P. M. & Churchland, P. S. (1990), ‘Could a machine think?’, *Scientific American* **262**, 26–31.

Clancy, M. J. & Linn, M. C. (1999), Patterns and pedagogy, *in* ‘The proceedings of the thirtieth SIGCSE technical symposium on Computer science education’, ACM Press, pp. 37–42.

Clark, A. (2004), ‘Embodiment and ecological control’.
URL: *http://www.stanford.edu/class/cs378/clark.ppt*

Clark, M. (1990), *Nietzsche on Truth and Philosophy - Modern European Philosophy*, Cambridge University Press, Cambridge.

Clausner, T. C. & Croft, W. (1997), ‘Productivity and schematicity in metaphors’, *Cognitive Science* **21**(3), 247–282.
URL: *www.cognitivesciencesociety.org/abstract/5-98clausner.html*

Cockburn, A. (2000), ‘Naur, ehn, musashi’.
URL: *At http://alistair.cockburn.us/crystal/books/asd/extracts/asdapp2/-asdapp2naurehnmusashi.htm*

Colbert, E. H. (1971), *Men and Dinosaurs: The Search in Field and Laboratory*, Pelican Books, Harmondsworth, Middlesex.

Conesa, J. (2005), ‘Uexküll’s umwelt and semiotic matrix theory’.
URL: *http://www.geocities.com/jorgeconesa/Biosemiotics/Uexkull.html#-abs*

Cooper, J. (2000), *Java Design Patterns: a Tutorial*, Addison-Wesley, Boston.

Cooperstein, M. A. (n.d.), ‘The conjoint evolution of creativity and consciousness: A developmental perspective’.
URL: *http://www.wynja.com/personality/candc.html*

Copeland, B. J. (2002), The Church-Turing Thesis, *in* E. N. Zalta, ed., 'The Stanford Encyclopedia of Philosophy'.
URL: *http://plato.stanford.edu/archives/fall2002/entries/church-turing/*

Coplien, J. (1996*a*), *Software Patterns*, SIGS Books, New York.

Coplien, J. (1996*b*), 'Space: The Final Frontier'. See reference to C++ Report.
URL: *C++Report/SpaceFinalFrontier-1.html*

Coplien, J. (1998), 'The Geometry of C++ Objects'. See reference to C++ Report.
URL: *C++Report/IdiomGeometry-1.html*

Coplien, J. (2000*a*), 'Close the Window and Put it On the Desktop'.
URL: *prog.vub.ac.be/˜imichiel/ecoop2000/workshop/subm_papers/coplien.pdf*

Coplien, J. (2000*b*), 'Multi-paradigm design'.
URL: *www.netobjectives.com/download/CoplienThesis.pdf*

Coplien, J. (2001), 'The Future of Language: Symmetry or Broken Symmetry?'.
URL: *In Proceedings of VS Live 2001, San Francisco, California, January 2001*

Coplien, J. (n.d.), 'A Pattern Definition'.
URL: *hillside.net/patterns/definition.html*

Coplien, J. & Zhao, L. (2000), Symmetry and symmetry breaking in software patterns, *in* 'Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering (GCSE'2000)', pp. 374–398.

Cox, R. & Brna, P. (1995), 'Supporting the use of external representations in problem solving: The need for flexible learning environments'.
URL: *citeseer.ist.psu.edu/cox95supporting.html*

Crawford, D. (1996), 'Editorial', *Communications of the ACM* **39**(10), 5.

Crosson, F. J. (1967), Information theory and phenomenology, *in* F. J. Crosson & K. M. Sayre, eds, 'Philosophy and Cybernetics', University of Notre Dame Press, Notre Dame, Indiana, pp. 99–136.

Curtis, B. (1982), A review of human factors research on programming languages and specifications, *in* 'Proceedings of the 1982 conference on Human factors in computing systems', ACM Press, New York, NY, USA, pp. 212–218.

Curtis, B. (1990), *Tutorial, Human Factors in Software Development*, IEEE Computer Society Press, Los Alamitos, CA, USA.

Daniels, M. Petre, M. & Berglund, A. (1998), 'Building a rigorous research agenda into changes in teaching', At http://www.docs.uu.se/docs/cse/papers/brisbane98.html.

Danzig, N. (n.d.), 'Recursion', At http://danzig.jct.ac.il/java_class/recursion.html.

D'Arcangelo, M. (2000), The scientist in the crib: A conversation with andrew meltzoff, *in* 'The Science of Learning', Vol. 58, Association for Supervision and Curriculum Development.
URL: *http://www.ascd.org/publications/ed_lead/200011/darcangelo.html*

Davies, P. (1989), *The Cosmic Blueprint*, Unwin Paperbacks, London.

Davies, S. (1993), Externalising information during coding activities: Effects of expertise, environment, and task, *in* C. R. Cook, J. C. Scholtz & J. C. Spohrer, eds, 'Empirical Studies of Programmers: Fifth Workshop', Ablex, Norwood, N.J., pp. 42–61.

Davies, S. P. (1990), 'The nature and development of programming plans', *International Journal of Man-Machine Studies* **32**(4), 461–481.

Davies, S. P. (1994), 'Knowledge restructuring and the acquisition of programming expertise', *International Journal of Human Computer Studies* **40**, 703–726.

Davis, S. B. & Moar, M. (2005), The amateur creator, *in* 'C&C '05: Proceedings of the 5th conference on Creativity & cognition', ACM Press, New York, NY, USA, pp. 158–165.

de Groot, A. D. (1965), *Thought and Choice in Chess*, Mouton, The Hague, the Netherlands.

de Groot, A. D. (1966), Perception and memory versus thought: Some old ideas and recent findings, *in* B. K, ed., 'Problem Solving: Research, Method and Theory', Wiley, New York.

Deek, F. & Friedman, R. (2002), Computing and composition: Common skills, common process, *in* 'Journal of Computer Science Education', Vol. 1, International Society for Technology in Education SIGCS, pp. 8–14.
URL: *http://d2.virt.pciwest.net/sigcs/community/jcseonline/2001/11/deek.html*

Delvin, K. (2000), *The Language of Mathematics*, Henry Holt and Co, New York.

DeNelsky, G. Y. & McKee, M. (2005), 'Prediction of computer programmer training and job performance using the aabp test', At http://www.waldentesting.com/about/article9.htm.

Dennet, D. (1978), *Brainstorms; Philosophical Essays of Mind and Psychology*, Bradford Books, Cambridge, Massachusetts.

Denning, P. J. (1989), 'A debate on teaching computing science', *Communications of the ACM* **32**(12), 1397–1414.

Devlin, K. (1995), *Logic and Information*, Cambridge University Press, Cambridge.

Dewey, J. (1966), *Democracy and Education : An Introduction to the Philosophy of Education*, Free Press, New York.

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J.

Dijkstra, E. W. (1982), *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York.

Discover Engineering (2005), 'Engineering: The stealth profession'.
URL: *http://www.discoverengineering.org/aboutengineers.asp*

Downes, S. M. (1992), 'The importance of models in theorizing: A deflationary semantic view', *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association* **1**, 142–153.

Draper, S. W. (2005), 'The hawthorne effect and other expectancy effects: a note'.
URL: *http://www.psy.gla.ac.uk/ steve/hawth.html*

Dreyfus, H. L. & Dreyfus, S. E. (n.d.), 'From socrates to expert systems: The limits and dangers of calculative rationality'.
URL: *http://ist-socrates.berkeley.edu/~hdreyfus/html/paper_socrates.html*

East, J. P. & Wallingford, E. (1997), Pattern-based programming in initial instruction (seminar), *in* 'Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education', ACM Press, p. 393.

Einstein, A. (1954), 'On Science'.

Einstein, A. (1962), *Relativity: The Special and General Theory*, Methuen, London.

Einstein, A. (1974), Moral Decay, *in* 'Out of my Later Years', The Citadel Press, Secaucus, New Jersey.

Eisner, E. W. (2000), Benjamin Bloom, *in* J. Hallak, ed., 'Prospects: the quarterly review of comparative education', Vol. XXX, UNESCO: International Bureau of Education, Paris.
URL: *http://www.ibe.unesco.org/International/Publications/Thinkers/-ThinkersPdf/bloome.pdf*

Elton, G. R. (1969), *The Practice of History*, Collins, London.

Evans, D. (2005), 'Transcript of interview on ABC radio program "All in the Mind" with Julie Browning on Saturday 26 November 2005'.
URL: *http://www.abc.net.au/rn/science/mind/stories/s1514225.htm*

Fabian, J. (1990), *Creative Thinking and Problem Solving*, Lewis Publishers, Inc, Chelsea, Michigan.

Fauconnier, G. (1999), 'Introduction to methods and generalizations'.
URL: *http://cogweb.ucla.edu/Abstracts/Fauconnier_99.html*

Feigenbaum, E. A. & McCorduck, P. (1983), *The fifth generation : artificial intelligence and Japan's computer challenge to the world*, Michael Joseph, London.

Ferguson, M. (1980), *The Aquarian Conspiricy*, J. P. Tarcher, Los Angeles.

Fernald, A. (1993), 'Approval and disapproval: Infant responsiveness to vocal affect in familiar and unfamiliar languages', *Child Development* **64**, 657–674.

Fincher, S. (1999*a*), What are we doing when we teach programming?, *in* 'Frontiers in Education '99', IEEE, pp. 12a41–5.
URL: *http://www.cs.kent.ac.uk/pubs/1999/917*

Fincher, S. (1999*b*), 'What is a Pattern Language?'.
URL: *http://www.hcipatterns.org/tiki-download$_f$ile.php?fileId* = 11

Fincher, S. & Utting, I. (2002), Pedagogical patterns: their place in the genre, *in* 'Proceedings of the 7th annual conference on Innovation and technology in computer science education', ACM Press, pp. 199–202.

Flanagan, O. (1991), *The Science of the Mind*, second edn, A Bradford Book, The MIT Press, Cambridge, Massachusetts.

Floyd, R. (1979), 'The paradigms of programming', *Communications of the ACM* **22**(8), 455–460.

Fodor, J. A. (1975), *The Language of Thought*, Thomas Y. Crowell Company, New York.

Fodor, J. A. (1998), *Concepts: Where Cognitive Science Went Wrong*, Clarendon Press, Oxford.

Forsee, A. (1963), *Albert Einstein: Theoretical Physicist*, Macmillan, New York.

Francois, W. (1964), *Industrialization comes of Age*, Collier-Macmillan Ltd., London.

Freeman, W. J. (1999), *How Brains Make Up Their Minds*, Phoenix Books, London.

French, M. (1994), *Invention and Evolution: Design in Nature and Engineering*, second edn, Cambridge University Press, Cambridge.

Gabriel, R. (1996*a*), Introduction-2: Repetition, generativity, and patterns, *in* J. M. Vlissides, J. O. Coplien & N. L. Kerth, eds, 'Pattern Languages of Program Design 2', Addison-Wesley, Reading, Massachusetts.

Gabriel, R. (1996*b*), *Patterns of Software: Tales from the Software Community*, Oxford University Press, New York.

Gagne, R. M. (1979), Learnable aspects of human thinking, *in* A. E. Lawson, ed., 'The Psychology of Teaching for Thinking and Creativity', ERIC Clearinghouse for Science, Mathematics, and Environmental Education, Ohio State University, Columbus, Ohio.

Gallagher, S. (2000), Self-reference and schizophrenia: A cognitive model of immunity to error through misidentification, *in* D. Zahavi, ed., 'Exploring the Self: Philosophical and Psychopathological Perspectives on Self-experience', John Benjamins, Amsterdam & Philadelphia.
URL: *http://www2.canisius.edu/ gallaghr/copenhagen.html*

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edn, Addison-Wesley, Reading, Mass.

Gannon, J. D. & Horning, J. J. (1975), The impact of language design on the production of reliable software, *in* 'Proceedings of the international conference on Reliable software', pp. 10–22.

Gardner, H. (1985), *The Mind's New Science*, Basic Books, New York.

Garner, S. (2001), Cognitive load reduction in problem solving domains, *in* 'Proceedings of the International Conference in Computer Education, ICCE2001'.

Garreau, J. (2005), 'Inventing our evolution'.
URL: *http://www.washingtonpost.com/wp-dyn/content/article/2005/05/15-/AR2005051501092.html*

Gelfand, N. et al. (1998), Teaching data structure patterns, *in* 'Proceedings of the ACM Symposium on Computer Science Education'.

Gellner, E. (1968), *Words and Things*, Penguin Books, Middlesex, England.

Geschwind, N. (1974), *Selected Papers on Language and the Brain*, D. Reidel, Dordrecht.

Gilbert, D. T. (1991), 'How mental systems believe', *American Psychologist* **46**, 107–119.

Gilbert, D. T., Trafarodi, R. W. & Malone, P. S. (1993), 'You can't not believe everything you read', *Journal of Personality and Social Psychology* **65**, 221–233.

Glass, R. L. (2002), 'Facts of software engineering management'.
URL: *http://www.awprofessional.com/articles/article.asp?p=30091&seqNum=5&rl=1*

Gleick, J. (1988), *Chaos : making a new science*, Heinemann, London.

Goertzel, B. (1993), *The Evolving Mind*, Gordon and Breach, Langhorne, Pa.

Gopnik, A. (1996), 'The scientist as child', *Philosophy of Science* **63**(4), 485–514.

Gopnik, A. (2003), 'Transcript of interview on ABC radio program "All in the Mind" with Natasha Mitchell on Sunday 3 August 2003'.
URL: *http://www.abc.net.au/rn/science/mind/index/s913231.htm*

Gopnik, A. & Meltzoff, A. (1997), *Words, Thoughts and Theories*, MIT Press, Cambridge, Massachusetts.

Grabow, S. (1983), *Christopher Alexander, The Search for a New Paradigm in Architecture*, Oriel Press, Stocksfield, Northumberland.

Graesser, A. C. & Clark, L. F. (1985), *Structures and Procedures of Implicit Knowledge*, Ablex Publishing Corporation, Norwood, New Jersey.

Green, T. (1997), 'Cognitive approaches to software comprehension'.
URL: *http://homepage.ntlworld.com/greenery/workStuff/Papers/LimerickTalk1997/LimerickTalk.html*

Greenberg, N. (2004*a*), 'Art and organism'.
URL: *http://notes.utk.edu/bio/greenberg.nsf*

Greenberg, N. (2004*b*), 'Truth in the brain: The neuroethology of belief'.
URL: *http://notes.utk.edu/bio/greenberg.nsf*

Greenfield, P. M. (1984), *Mind and Media*, Harvard University Press, Cambridge, Massachusetts.

Groundwater-Smith, S., Brennan, M., McFadden, M. & Mitchell, J. (2001), *Schooling in a Changing World*, Harcourt Australia, Sydney.

Gruenberger, F. (1977), 'So you're trying to teach computing', *Datamation* **23**(4), 119–124.

Guisepi, R. A. (n.d.), 'The history of education'.
URL: *http://ragz-international.com/history_of_education.htm*

Hall, E. T. (1976), *Beyond Culture*, Doubleday, New York.

Hall, W. P. (2005), 'Biological nature of knowledge in the learning organization', *The Learning Organization* **12**(2), 169–188.
URL: *http://informatics.indiana.edu/rocha/emcsr96.html*

Halldén, S. (1997), Creativity and the evolutionary viewpoint, *in* A. E. Andersson & N.-E. Sahlin, eds, 'The Complexity of Creativity', Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 5–16.

Halpern, D. F. (1987), Analogies as a critical thinking skill, *in* D. E. Berger, K. Pezdek & W. P. Banks, eds, 'Applications of cognitive psychology : problem solving, education, and computing', L. Erlbaum Associates, Hillsdale, N.J.

Harley, T. (2001), *The Psychology of Language: From Data to Theory*, 2nd edn, Psychology Press, Belmont, California.

Harrison, N. (2001), Teaching patterns: Patterns for teaching patterns in a classroom setting, *in* 'Pattern Languages of Programs, Second Asian Pacific Conference', Pattern Languages of Programs, Second Asian Pacific Conference.

Harrison, W. A. & Magel, K. I. (1981), A suggested course in introductory computer programming, *in* 'Twelfth SIGCSE Technical Symposium on Computer Science Education', pp. 50–56.

Harth, E., ed. (1993), *The Creative Loop: How the Brain makes a Mind*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Hauck, B. B. & Freehill, M. F. (1972), *The Gifted - Case Studies*, Wm. C. Brown Company, Dubuque, Iowa.

Haugeland, J. (1985), *Artificial Intelligence: The Very Idea*, MIT Press, Cambridge, Massachusetts.

Haverty, L. A., Koedinger, K. R., Klahr, D. & Alibali, M. W. (2000), 'Solving Inductive Reasoning Problems in Mathematics', *Cognitive Science* **24**(2).
URL: *www.cognitivesciencesociety.org/abstract/haverty.html*

Haviland, R. M. & Lelwica, M. (1987), 'The induced affect response: 10-week-old infants' responses to three emotion expressions', *Developmental Psychology* **23**(1), 97–104.

Heidegger, M. (1962), *Being and Time*, Harper & Row, New York.

Heims, S. J. (1996), *Constructing a Social Science for Postwar America. The Cybernetics Group, 1946-1953*, MIT, Cambridge, Massachussetts.

Henry, C. & Rocha, L. M. (1996), 'Language theory:consensual selection of dynamics', *Cybernetics and Systems: An International Journal* **27**, 541–553.
URL: *http://informatics.indiana.edu/rocha/emcsr96.html*

Herbert, J. (1997), 'The tasks of programming'.
URL: *educ.queensu.ca/~brownan/courses/aqcsdp97fall/tasksofprog.htm*

Heylighen, F. (1996), 'What is complexity?'.
URL: *http://pespmc1.vub.ac.be/COMPLEXI.html*

Hillside (2005), 'Patterns library'.
URL: *http://www.hillside.net/patterns/*

Hiltz, S. R. & Turoff, M. (2005), 'Education goes digital: the evolution of online learning and the revolution in higher education', *Communications of the ACM* **48**(10), 59–64.

Hinds, P. J., Patterson, M. & Pfeffer, J. (2001), 'Bothered by abstraction: The effect of expertise on knowledge transfer and subsequent novice performance', *Journal of Applied Psychology* **86**(6), 1232–1243.

Hoare, C. A. R. (1973), 'Hints on programming language design'.
URL: *http://www.eecs.umich.edu/ bchandra/courses/papers/HoareHints.-pdf*

Hoffman, B. (1972), *Albert Einstein: Creator and Rebel*, Viking Press, New York.

Hofstadter, D. R. (1979), *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York.

Hofstadter, D. R. (1985), *Metamagical Themas: Questing for the Essence of Mind and Pattern*, Penguin Books, London.

Holland, J. (1996), *Hidden Order, How Adapatation Builds Complexity*, Perseus Press, Philadelphia.

Holmboe, C. (1999), A cognitive framework for knowledge in informatics: the case of object-orientation, *in* 'Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education', ACM Press, pp. 17–20.

Holzman, T. G., Pellegrino, J. W. & Glaser, R. (1982), 'Cognitive dimensions of numerical rule induction', *Journal of Educational Psychology* **74**, 360–373.

Hopkins, M. & DuBois, C. (2005), 'New software can help people make better decisions in time-stressed situations'.
URL: *http://live.psu.edu/index.php?sec=vs&story=12894&pf=1*

Horowitz, E. (1983), *Fundamentals of Programming Languages*, Computer Science Press, Rockville, Maryland.

Hume, D. (1740), *A Treatise of Human Nature*, University of Adelaide, Adelaide, South Australia.
URL: *http://etext.library.adelaide.edu.au/h/h92t/chapter34.html*

Johnson-Laird, P. N. (1993), *Human and Machine Thinking*, Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey.

Johnson, M. (1987), *The Body in the Mind*, University of Chicago Press, Chicago.

Joseph, R. (1980), 'Awareness, the origin of thought, and the role of conscious self-deception in resistance and repression', *Psychological Reports* **46**, 767–781.

Joseph, R. (1982), 'The neuropsychology of development: Hemispheric laterality, limbic language, and the origin of thought', *Clinical Psychology* **44**, 4–33.

Joseph, R. (n.d.), 'Language, consciousness and the origin of thought'.
URL: *http://Brain-Mind.com/languageThought.html*

jt (2000), 'Godel's theorem', At http://www.everything2.net/index.pl?node=Godel's%20Theorem.

Kaipa, P. & Johnson, S. (1999), 'Igniting your Natural Genius - Section 1: Instinctive Learning'.
URL: *http://www.mithya.com/learning/igniting01.html*

Kaiser, K. M. (1985), 'The relationship of cognitive style to the derivation of information requirements', *SIGCPR Comput. Pers.* **10**(2), 2–12.

Kant, I. (1881), *Critique of Pure Reason*, Macmillan, London.

Kapor, M. (1996), A software design manifesto, *in* T. Winograd, ed., 'Bringing Design to Software', ACM Press, New York.

Kay, A. C. (1993), The early history of smalltalk, *in* 'The second ACM SIGPLAN conference on History of programming languages', ACM Press, pp. 69–95.

Khamsi, R. (2004), 'Electrical brainstorms busted as source of ghosts'.
URL: *http://www.nature.com/news/2004/041206/pf/041206-10$_p$f.html*

Kidder, T. (1981), *The Soul of a New Machine*, Penguin Books, Middlesex, England.

Klotz, J. (2004), 'Intellectual disability: Communicating across the divide'.
URL: *http://www.abc.net.au/rn/science/mind/stories/s1261362.html*

Knuth, D. E. (1984), 'Literate programming', *The Computer Journal* **27**(2), 97–111.

Kodituwakku, S. R. & Bertok, P. (2003), Pattern categories: A mathematical approach for organizing design patterns, *in* J. Noble, ed., 'Pattern Languages of Programs 2002. Revised papers from the Third Asia-Pacific Conference on Pattern Languages of Programs, (KoalaPLoP 2002)', Vol. 13 of *Conferences in Research and Practice in Information Technology*, ACS, Melbourne, Australia, p. 63.

Kölling, M. (2005), 'BlueJ - The Interactive Java Environment'.
URL: *http://www.bluej.org/*

Koubek, R. J., Salvendy, G., Dunsmore, H. E. & LeBold, W. K. (1989), 'Cognitive issues in the process of software development: review and reappraisal', *International Journal of Man-Machine Studies* **30**, 171–191.

Kremer, R. (1997), 'Constraint graphs: A concept map meta-language'.
URL: *http://pages.cpsc.ucalgary.ca/ kremer/dissertation/Ch1.ps*

Kroeber, A. L. (1923), *Anthropology: Culture Patterns & Processes*, Harbinger Books, New York.

Krotoski, A. (2005), 'Game for learning'.
URL: *http://www.technologyreview.com/articles/05/04/wo/wo_040605krotoski.asp*

Krueger, C. W. (1992), 'Software reuse', *ACM Computing Surveys (CSUR)* **24**(2), 131–183.

Kuhn, T. (1962), *The Structure of Scientific Revolutions*, University of Chicago Press, Chicago.

Kyllonen, P. C. & Christal, R. E. (1990), 'Reasoning ability is (little more than) working-memory capacity?!', *Intelligence* **14**, 389–433.

Laing, R. D. (1973), *The Politics of Experience and The Bird of Paradise*, Penguin Books Ltd, Middlesex.

Lakoff, G. (1993), The contemporary theory of metaphor, *in* A. Ortony, ed., 'Metaphor and Thought', second edn, Cambridge University Press, Cambridge.

Lakoff, G. & Johnson, M. (1999), *Philosophy in the Flesh*, Basic Books, New York.

Lakoff, G. & Nunez, R. E. (2000), *Where mathematics comes from : how the embodied mind brings mathematics into being*, Basic Books, New York.

Lammers, T. G. (2005), 'Identification of plants'.
URL: *http://www.bookrags.com/sciences/biology/identification-of-plants-plsc-03.html*

Langer, S. K. (1962), *Philosophical Sketches*, Oxford University Press, London.

Langer, S. K. (1976), *Philosophy in a New Key*, Harvard University Press, Cambridge, Massachusetts.

Lea, D. (1993), 'Christopher Alexander: An Introduction for Object-Oriented Designers'.
URL: *gee.cs.oswego.edu/dl/ca/ca/ca.html*

Lee, J. M. & Shneiderman, B. (1978), Personality and programming: Time-sharing vs. batch preference, *in* 'Proceedings of the 1978 annual conference', ACM Press, pp. 561–569.

Lee, R. (2003), 'Java For Non-Programmers'.
URL: *http://ryanlee.org/tutorials/java/beginners*

Lesgold, A., Rubinson, H., Feltovich, P., Glaserand, R., Klopfer, D. & Wang, Y. (1988), Expertise in a complex skill: Diagnosing x-ray pictures, *in* M. T. H. Chi, R. Glaser & M. J. Farr, eds, 'The nature of expertise', Erlbaum, Hillsdale, NJ.

Levi, I. (2004), 'The logic of consistency and the logic of truth', *Dialectica* **58**(4), 461–482.

Lewin, K. (1951), *Field Theory in Social Science*, Harper & Rowe, New York.

Lewis, C. (1981), Skill in Algebra, *in* J. R. Anderson, ed., 'Cognitive Skills and their Acquisition', Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 85–110.

Liddle, D. (1996), An interview with david liddle, *in* T. Winograd, ed., 'Bringing Design to Software', ACM Press, New York.

Lister, R. & Leaney, J. (2003), First year programming: Let all the flowers bloom, *in* T. Greening & R. Lister, eds, 'Fifth Australasian Computing Education Conference (ACE2003)', Vol. 20 of *Conferences in Research and Practice in Information Technology*, ACS, Adelaide, Australia, pp. 221–230.

Locke, J. (1910), *An Essay Concerning Human Understanding*, new revised edition edn, Routledge, London.

Lockheed, J. (1979), An introduction to cognitive process instruction, *in* J. Lochhead & J. Clement, eds, 'Cognitive Process Instruction', Franklin Institute Press, Philadelphia, Pa.

Lucas, H. C. & Kaplan, R. B. (1976), 'A structured programming experiment.', *The Computer Journal* **19**(2), 136–138.

Luriia, A. R. (1973), *The working brain; an introduction to neuropsychology*, Basic Books, New York.

M. E. Sime, T. R. G. Green, D. J. G. (1977), 'Scope marking in computer conditionals - a psychological evaluation', *International Journal of Man-Machine Studies* **9**, 107–118.

Macnab, D. S. (2000), 'Forces for change in mathematics education: The case of timss!', *Education Policy Analysis Archives* **8**(15).
URL: *http://epaa.asu.edu/epaa/v8n15.html*

Mahood, J. & Richards, J. (2000), 'Gender and assessment in mathematics', At http://www.geocities.com/mathladies/gender/gender.html.

Maj, S. P., Veal, D. & Charlesworth, P. (2000), Is computer technology taught upside down?, *in* '5th annual SIGCSE/SIGCUE conference on Innovation and technology in computer science education', ACM Press, pp. 140–143.

Martin, E. W. & Badre, A. N. (1977), Problem formulation for programmers, *in* 'Proceedings of the seventh SIGCSE technical symposium on Computer science education', ACM Press, pp. 133–138.

Massey, J. L. (1967), Information, machines, and men, *in* F. J. Crosson & K. M. Sayre, eds, 'Philosophy and Cybernetics', University of Notre Dame Press, Notre Dame, Indiana.

Matlin, M. W. (1994), *Cognition*, Harcourt Brace College Publishers, Fort Worth, Texas.

Mattson, S. (1996), *Object oriented frameworks: a Survey and Methodological Issues*, Lund University, Lund, Sweden.

Maturana, H. R. (1970), Biology of cognition. biological computer laboratory research report, bcl 9.0. university of illinois, urbana, *in* H. R. Maturana & F. J. Varela, eds, 'Autopoiesis and Cognition: The realization of the Living', Reidel Publishing Co., Dordrecht, The Netherlands, pp. 5–58.
URL: *http://www.enolagaia.com/M70-80BoC.html#I*

Mayer, D. B. & Stalnaker, A. W. (1968), Selection and evaluation of computer personnel- the research history of sig/cpr, *in* 'Proceedings of the 1968 23rd ACM national conference', ACM Press, New York, NY, USA, pp. 657–670.

Mayer, R. (n.d.), The elusive search for teachable aspects of problem solving, *in* I. Glover & R. Ronning, eds, 'Cognitive Process Instruction', Academic Press, New York.

Mayer, R. E. (1981), 'The psychology of how novices learn computer programming', *ACM Comput. Surv.* **13**(1), 121–141.

Mayer, R. E. (1985), 'Learning in complex domains: A cognitive analysis of computer programming', *Psychology of Learning and Motivation* **19**, 89–130.

Mayer, R. E., Dyck, J. L. & Vilberg, W. (1986), 'Learning to program and learning to think: what's the connection?', *Communications of the ACM* **29**(7), 605–610.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001), 'A multi-national, multi-institutional study of assessment of programming skills of first-year cs students', *SIGCSE Bulletin* **33**(4), 125–180.

McCullock, W. S. (1965), *Embodiments of Mind*, MIT Press, Cambridge, Massachusetts.

McIllwain, D. (2005), 'Transcript of interview on ABC radio program "All in the Mind" with Julie Browning on Saturday 26 November 2005'.
URL: *http://www.abc.net.au/rn/science/mind/stories/s1514225.htm*

McKeown, J. et al. (1999), 'Why we need to develop success in introductory programming courses'.
URL: *http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html*

McPeck, J. E. (1981), *Critical Thinking and Education*, St. Martin's Press, New York.

Meek, B., Heath, P. & Rushby, N. (1983), *Guide to Good Programming Practice*, 2nd edn, Ellis Horwood Ltd, Chichester, West Sussex.

Mill, J. S. (1946), *Four Dialogues of Plato: Including the Apology of Socrates*, C.A. Watts & Co, Ltd., London.

Mill, J. S. (1981), *Autobiography and literary essays*, University of Toronto Press, Toronto.

Miller, G. A. (1956), 'The magical number seven, plus or minus two: Some limits on our capacity to process information', *Psychological Review* **63**, 81–97.

Monaghan, P. (1995), 'A corpus-based analysis of individual differences in proof-style'.
URL: *citeseer.ist.psu.edu/monaghan95corpusbased.html*

Monod, J. (1974), *Chance and Necessity*, Fontana Books, Glasgow.

Morris, D. (1991), *Babywatching*, Jonathon Cape, London.

Mott, M. (2003), 'Can animals sense earthquakes?'.
URL: *http://news.nationalgeographic.com/news/2003/11/1111_031111_earthquakeanimals.html*

Mulholland, P. & Eisenstadt, M. (2002), 'Chapter x: Using software to teach computer programming: Past, present and future'.
URL: *http://www.kmi.open.ac.uk/people/paulm/sv-papers/using.ps*

Nassi, I. & Shneiderman, B. (1973), ‘Flowchart techniques for structured programming’, *SIGPLAN Not.* **8**(8), 12–26.

Naur, P. (1985), ‘Programming as theory building’, *Microprocessing and Microprogramming* **15**, 253–261.

Nelson, J. (1999), *Programming mobile objects with Java*, Wiley, New York.

Neves, D. M. & Anderson, J. R. (1981), Knowledge Compilation: Mechanisms for the Automatization of Cognitive Skills, *in* J. R. Anderson, ed., ‘Cognitive Skills and their Acquisition’, Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 57–84.

Newell, A. & Rosenbloom, P. S. (1981), Mechanisms of Skill Acquisition and the Law of Practice, *in* J. R. Anderson, ed., ‘Cognitive Skills and their Acquisition’, Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 1–55.

Newell, A., Shaw, J. C. & Simon, H. A. (1963), Empirical explorations with the logic theory machine: A case study in heuristics, *in* E. A. Feigenbaum & J. Feldman, eds, ‘Computers and Thought’, McGraw-Hill, New York, pp. 109–133.

Newsted, P. R. (1975), ‘Grade and ability predictions in an introductory programming course’, *SIGCSE Bull.* **7**(2), 87–91.

Nickerson, R. S., Perkins, D. N. & Smith, E. E. (1985), *The Teaching of Thinking*, Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey.

Northrop, L. M. (1992), Finding an educational perspective for object-oriented development, *in* ‘Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)’, ACM Press, pp. 245–249.

Novak, J. D. (1977), *A Theory of Education*, Cornell University Press, New York.

Novak, J. D. (2004), ‘A Science Education Research Program that Led to the Development of the Concept Mapping Tool and a New Model for Education’.
URL: *http://www.cmc.ihmc.us/papers/cmc2004-286.pdf*

Novik, L. R. (1990), ‘Analogical Transfer, Problem Similarity, and Expertise’, *Journal of Experimental Psychology: Learning, Memory, and Cognition* **14**(3), 510–520.

Nummedal, S. G. (1987), Developing reasoning skills in college students, *in* D. E. Berger, K. Pezdek & W. P. Banks, eds, ‘Applications of cognitive psychology : problem solving, education, and computing’, L. Erlbaum Associates, Hillsdale, N.J.

Odenwald, S. (2002), ‘Why nothing is important: Review of patterns in the void’.
URL: *http://www.astronomycafe.net/vacuum/vactext.html*

Odenwald, S. (2004), ‘Ask the Physicist’.
URL: *http://einstein.stanford.edu/content/relativity/qanda.html*

Ortony, A. (1979), Metaphor: A multidimensional problem, *in* A. Ortony, ed., ‘Metaphor and Thought’, first edn, Cambridge University Press, Cambridge.

Oxford Dictionary (2005), At http://dictionary.oed.com.

Paas, F., Renkl, A. & Sweller, J. (2003), ‘Cognitive load theory and instructional design: Recent developments’, *Educational Psychologist* **38**, 1–4.

Papert, S. (1980), *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, New York.

Pattee, H. H. (1969), ‘How does a molecule become a message?’, *Developmental Biology Supplement* **3**, 1–16.

Pattee, H. H. (1996), The physics of symbols and the evolution of semiotic controls, *in* ‘Santa Fe Institute Studies in the Sciences of Complexity’, Addison-Wesley, Redwood City, CA.
URL: *www.ws.binghamton.edu/pattee/semiotic.html*

Pattee, H. H. (2001*a*), ‘Irreducible and complementary semiotic forms’, *Semiotica* **134**, 341–358.

Pattee, H. H. (2001*b*), ‘The physics of symbols: Bridging the epistemic cut’, *Biosystems* **60**, 5–21.
URL: *http://informatics.indiana.edu/rocha/pattee/pattee.html*

Penfield, W. (1975), *The Mystery of the Mind: A Critical Study of Consciousness and the Human Brain*, Princeton University Press, Princeton, New Jersey.

Pennington, N. (1987), Comprehension strategies in programming, *in* G. M. Olson, S. Sheppard & E. Soloway, eds, ‘Empirical Studies of Programmers: Second Workshop’, Ablex, Norwood, N.J.

Penrose, R. (1989), *The Emporer’s New Mind*, Oxford University Press, London.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989), Conditions of learning in novice programmers, *in* E. Soloway & J. C. Spohrer, eds, ‘Studying the novice programmer’, Ablex Corp., Norwood, New Jersey, pp. 261–279.

Perkins, D. N., Schwartz, S. & Simmonds, R. (1988), Instructional strategies for the problems of novice programmers, *in* R. E. Mayer, ed., ‘Teaching and Learning Computer Programming: Multiple Research Perspective’, Erlbaum, Hillsdale, New Jersey, pp. 153–178.

Perlis, A. J. (1982), ‘Epigrams on programming’, *SIGPLAN Notices* **17**(9).

Pezdek, K. (1987), Television comprehension as an example of applied research in cognitive psychology, *in* D. E. Berger, K. Pezdek & W. P. Banks, eds, ‘Applications of cognitive psychology : problem solving, education, and computing’, L. Erlbaum Associates, Hillsdale, N.J.

Pfleeger, S. L. (1998), *Software Engineering: Theory and Practice*, Prentice Hall, Upper Saddle River, NJ.

Piaget, J. (1952), *The Origins of Intelligence in Children*, International University Press, New York.

Piaget, J. (1972), *Insights and Illusions of Philosophy*, Routledge & Kegan Paul Ltd, London.

Piaget, J. & Szeminska, A. (1952), *The Child’s Conception of Number*, Humanities Press, New York.

Pinker, S. (1994), *The Language Instinct*, Penguin Books, London.

Pinker, S. (1997), *How the Mind Works*, The Softback Preview, London.

Plank, W. G. (1998), ‘The Quantum Nietzsche’.
URL: *www.msubillings.edu/modlang/bplank/qn71_80.html*

Plato (1999), ‘Phaedo’.
URL: *http://etext.library.adelaide.edu.au/aut/phaedo.html*

Polanyi, M. (1958), *Personal Knowledge: Towards a Post-Critical Philosophy*, Routledge & Kegan Paul, London.

Polya, G. (1948), *How To Solve It: A New Aspect of Mathematical Method*, Princeton University Press, New Jersey.

Popper, K. (1959), *The Logic of Scientific Discovery*, Basic Books, New York.

Popper, K. (1966), *The Open Society and its Enemies*, Vol. 2, Routledge & Kegan Paul Ltd., London.

Popper, K. (1972), *Objective Knowledge: An Evolutionary Approach*, Oxford University Press, London.

Popper, K. (1977), *The Self and It’s Brain*, Routledge & Kegan Paul Ltd., London.

Porter, R. & Calder, P. (2003*a*), Applying patterns to novice programming problems, *in* J. Noble, ed., ‘Pattern Languages of Programs 2002. Revised papers from the Third Asia-Pacific Conference on Pattern Languages of Programs, (KoalaPLoP 2002)’, Vol. 13 of *Conferences in Research and Practice in Information Technology*, ACS, Melbourne, Australia, p. 73.

Porter, R. & Calder, P. (2003*b*), A pattern-based problem-solving process for novice programmers, *in* T. Greening & R. Lister, eds, 'Fifth Australasian Computing Education Conference (ACE2003)', Vol. 20 of *Conferences in Research and Practice in Information Technology*, ACS, Adelaide, Australia, pp. 231–238.

Porter, R. & Calder, P. (2004), Patterns in learning to program - an experiment?, *in* R. Lister & A. Young, eds, 'Sixth Australasian Computing Education Conference (ACE2004)', Vol. 30 of *Conferences in Research and Practice in Information Technology*, ACS, Dunedin, New Zealand, pp. 231–238.

Porter, R., Coplien, J. O. & Winn, T. (2005), 'Sequences as a basis for pattern language composition', *Science of Computer Programming* **56**(1-2), 231–249.

Preiss, B. (1999), Design patterns for the data structure and algorithms course, *in* 'Proceedings of the SIGCSE Technical Symposium', pp. 95–99.

Prensky, M. (2001), *Digital Game-Based Learning*, McGraw-Hill, New York.

Pressman, R. (1997), *Software Engineering: A Practitioner's Approach*, 4th edn, McGraw Hill, New York.

Pribram, K. H. (1971), *Languages of the Brain: Experimental Paradoxes and Priciples in Neuropschology*, Prentice-Hall, New Jersey.

Proulx, V. K. (2000), Programming patterns and design patterns in the introductory computer science course, *in* 'Proceedings of the thirty-first SIGCSE technical symposium on Computer science education', ACM Press, pp. 80–84.

Psychometrics Inc (2005), 'Which test should we use?', At http://www.psy-test.com/TestIndex.html.

Quine, W. V. O. (1966), *The Ways of Paradox and Other Essays*, Random House, Inc., New York.

Ramachandran, V. S. & Blakeslee, S. (1999), *Phantoms in the Brain : Probing the Mysteries of the Human Mind*, Harper Collins, New York.

Ramalingam, V. & Weidenbeck, S. (1997), 'An empirical study of novice program comprehension, in the imperative and object-oriented styles'.
URL: *http://www.cs.duke.edu/education/courses/fall00/cps189s/readings/-p124-ramalingam.pdf*

Rand, A. (1982), *Philosophy, Who Needs It*, Signet, New York.

Rayside, D. & Campbell, G. T. (2000), Aristotle and object-oriented programming: why modern students need traditional logic, *in* 'Proceedings of the thirty-first SIGCSE technical symposium on Computer science education', ACM Press, pp. 237–244.

Recio, A. M. & Godino, J. D. (2001), 'Institutional and personal meanings of proof', *Educational Studies in Mathematics* **48**(1), 83–99.

Reed, D. (1998), Incorporating problem-solving patterns in cs1, *in* 'Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education', ACM Press, pp. 6–9.

Renkl, A. (1997), 'Learning from worked-out examples: A study on individual differences', *Cognitive Science* **21**(1), 1–29.
URL: *www.cognitivesciencesociety.org/abstract/5-98renkl.html*

Riley, D. D. (2002), *The Object of Java*, Addison Wesley, Boston.

Rippa, S. A. (1967), *Education in a free society; an American history*, D. McKay Co., New York.

Rising, L. (1996), 'Patterns: Elements of reusable architectures'.
URL: *http://members.cox.net/risingl1/articles/patterns.htm*

Rising, L. (1997), 'The Road, Christopher Alexander, and Good Software Design'.
URL: *http://members.cox.net/risingl1/articles/goodsoft.htm*

Roberts, G. (2000), *COMP1102 Computer Programming 1, Semester 2, 2000*, Filnders University, Bedford Park.

Rubinstein, M. F. (1975), *Patterns of Problem Solving*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey.

Ruiz-Primo, M. A. (2004), 'Examining concept maps as an assessment tool'.
URL: *http://www.cmc.ihmc.us/papers/cmc2004-036.pdf*

Russell, B. (1961), Can men be rational?, *in* 'Let the People Think', The Rationalist Press Association Ltd, London.

Sackman, H. (1970), *Man-Computer Problem Solving*, Auerback Publishers, Princeton, New Jersey.

Salingaros, N. (1998), 'The structure of pattern languages'.
URL: *www.math.utsa.edu/sphere/salingar/ StructurePattern.html*

Salingaros, N. (2002*a*), 'Christopher Alexander's 'The Nature of Order''.
URL: *www.math.utsa.edu/sphere/salingar/NatureofOrder.html*

Salingaros, N. (2002*b*), 'Some Notes on Christopher Alexander'.
URL: *www.math.utsa.edu/sphere/salingar/Chris.text.html*

Sapir, E. (1971), *Language*, Rupert Hart-Davis, London.

Sargent, P. (1994), 'Design science or non-science: A non-scientific theory of design', *Design Studies* **15**(4), 389–402.
URL: *http://home.klebos.net/philip.sargent/design/design-studies.html*

Scanlan, D. A. (1988), The mental abilities associated with programming aptitude, *in* 'CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science', ACM Press, New York, NY, USA, p. 737.

Scherer, K. R. (1984), on the nature and function of emotion: A component process approach, *in* K. R. Scherer & P. Ekman, eds, 'Approaches to Emotion', Erlbaum, Hillsdale, New Jersey, pp. 293–317.

Schoenemann, P. T. (1999), 'Syntax as an emergent characteristic of the evolution of semantic complexity', *Minds and Machines* **9**, 309–346.
URL: *http://www.isrl.uiuc.edu/ amag/langev/paper/schoenemann99syntax-As.html*

Searle, J. R. (1990), Is the brain a digital computer?, *in* 'Proceeding of the 1990 conference of American Philosophical Association', Vol. 64, APA, pp. 21–37.

Secada, W. G. & Carey, D. A. (1990), 'Teaching mathematics with understanding to limited english proficiency students'.
URL: *http://www.wcer.wisc.edu/ccvi/pub/manuscript/Secada– Teaching _Math_To_LEP_Students.pdf*

Service, I. N. (2005), 'Sony researchers create 'curious' aibos'.
URL: *http://www.itworld.com/Tech/2987/050614aibo/pfindex.html*

Shalloway, A. (2003), 'Can patterns be harmful?', *Cutter IT Journal* **16**(9), 10–16.

Sheard, J. & Hagan, D. (1998), Our failing students: a study of a repeat group, *in* 'Proceedings of the 6th annual conference on the teaching of computing/3rd annual conference on integrating technology into computer science education on Changing the delivery of computer science education', ACM Press, pp. 223–227.

Sheffield Hallam University (2002), 'Visual ethnography of the hidden curriculum in higher education'.
URL: *http://www.shu.ac.uk/cgi-bin/news_full.pl?id_num=PR299&db=02*

Sheil, B. A. (1981), 'The psychological study of programming', *ACM Computing Surveys* **13**(1), 101–120.

Sheppard, S., Curtis, B., Milliman, P. & Love, T. (1979), 'Modern coding practices and programmer performance', *Computer* **12**, 41–49.

Shiffrin, R. M. (1976), Capacity limitations in information processing, attention, and memory, *in* W. K. Estes, ed., 'Handbook of learning and cognitive process', Vol. 4, Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 177–236.

Shiffrin, R. M. & Dumais, S. T. (1981), The Development of Automatism, *in* J. R. Anderson, ed., 'Cognitive Skills and their Acquisition', Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 111–140.

Shneiderman, B. (1976*a*), 'Exploratory experiments in programmer behavior', *International Journal of Computer Information Science* **5**, 123–143.

Shneiderman, B. (1976*b*), 'Exploratory experiments in programmer behavior', *International Journal of Computer & Information Science* **5**(2), 123–143.

Shneiderman, B. (1976*c*), 'Exploratory experiments in programmer behavior', *International Journal of Computer and Information Sciences* **5**(2), 123–143.

Shneiderman, B. (1980), *Software Psychology*, Winthrop, Cambridge, Massachussetts.

Shneiderman, B., Mayer, R., McKay, D. & Heller, P. (1977), 'Experimental investigations of the utility of detailed flowcharts in programming', *Commun. ACM* **20**(6), 373–381.

Shukla, M. (n.d.), 'Fermi's (non) Discovery of Nuclear Fission'.
URL: *http://www.geocities.com/madhukar_shukla/crebook/28.html*

Sime, M. E., Green, T. R. G. & Guest, D. J. (1973), 'Psychological evaluation of two conditional constructions used in computer languages', *International Journal of Man-Machine Studies* **5**, 105–113.

Simon, H. & Gilmartin, K. (1973), 'A simulation of memory for chess positions', *Cognitive Psychology* **5**, 29–46.

Skemp, R. (1971), *The Psychology of Learning Mathematics*, Penguin Books, Middlesex.

Skinner, B. F. (1953), *Science and Human Behaviour*, Macmillan, New York.

Sleeman, D. (1986), 'The challenges of teaching computer programming', *Communications of the ACM* **29**(9), 840–841.

Smith, C. (1981), *A Search for Structure: Selected Essays on Science, Art, and History*, The MIT Press, Cambridge, Massachusetts.

Smith, T. H. (2003), 'Metaphor in Mediation'.
URL: *www.metaresolution.com/Metaphor/web_axonfiles/mainmenu.htm*

Soloway, E. (1986), 'Learning to program = learning to construct mechanisms and explanations', *Communications of the ACM* **29**(9), 850–858.

Soloway, E. & Spohrer, J. C., eds (1989), *Studying the novice programmer*, Ablex Corp., Norwood, New Jersey.

Somogyi, S. (1999), 'Object-Oriented Basic for the Mac', *Macworld* .
URL: *http://www.macworld.com/1999/10/reviews/realbasic/*

Spinney, L. (2005), 'Artificial intelligence marches forward', *Scientist* **19**(5).

Stewart, I. (1995), *Nature's Numbers: Discovering Order and Pattern in the Universe*, Weidenfeld & Nicolson, London.

Taggart, B. (2000), 'From novice to expert'.
URL: *http://www.the-intuitive-self.org/scripts/frameit/author.cgi?/website-/author/memoir/threads/business_rationality.html*

Taylor, A. J. P. (1964), *The Origins of the Second World War*, Penguin Books, Middlesex, England.

Taylor, R. N. & Benbasat, I. (1980), A critique of cognitive styles theory and research, *in* 'Proceedings of the First International Conference on Information Systems'.

Tekinerdogan, B. & Aksit, M. (1999), 'On the notion of software engineering: A problem solving perspective'.
URL: *citeseer.nj.nec.com/434350.html*

Thorpe, W. H. (1962), *Biology and the Nature of Man*, Oxford University Press, London.

Tinbergen, N. (1951), *The Study of Instinct*, Clarendon Press, Oxford.

Tracy, H. H. (2003), 'Structured algorithm charts'.
URL: *http://faculty.frostburg.edu/cosc/htracy/cosc390/LM4SADinJAVA/-SAC.htm*

Tulving, E. (1972), Episodic and semantic memory, *in* E. Tulving & W. Donaldson, eds, 'Organization of memory', Academic Press, New York.

Tulving, E. & Thompson, D. M. (1973), 'Encoding specificity and retrieval processes in episodic memory', *Psychological Review* **80**, 352–373.

Turkle, S. & Papert, S. (1992), 'Epistemological pluralism and the revaluation of the concrete', *Journal of Mathematical Behaviour* **11**(1), 3–33.
URL: *http://www.papert.org/articles/EpistemologicalPluralism.html*

Turski, W. M. (1979), Look ahead at software engineering, *in* 'ICSE '79: Proceedings of the 4th international conference on Software engineering', IEEE Press, pp. 449–456.

Van Gog, T., Paas, F. & Merriënboer, J. J. G. V. (2004), 'Recommendations for research on task formats that model expert approaches to problem solving'.
URL: *www.iwm-kmrc.de/workshops/sim2004/pdf_files/VanGog_et_al.pdf*

VanLengen, C. & Maddux, C. (1990), 'Does instruction in computer programming improve problem solving ability', *Journal of IS Education* **2**(2), 47–49.

Venners, B. (2002), 'Object design workshop: Designing with patterns'.
URL: *www.artima.com/javaseminars/modules/DesPatterns/*

Vygotskii, L. S. (1962), *Thought and language*, Cambridge : M.I.T. Press.

Waddel, K. C. & Cross, J. H. (1988), Survey of empirical studies of graphical representations for algorithms, *in* 'CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science', ACM Press, New York, NY, USA, p. 696.

Waisvisz, M. (2004), 'Steim (studio for electro-instrumental music) foundation'.
URL: *http://www.steim.org/steim/info.html*

Walle, J. A. V. D. (2004), *Elementary and Middle School Mathematics, Teaching Developmentally*, Allyn & Bacon, Boston, MA.

Wallingford, E. (1996), Toward a first course based on object-oriented patterns, *in* 'SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education', ACM Press, New York, NY, USA, pp. 27–31.

Wallingford, E. (1998), 'Roundabout: A pattern language for recursive programming'.
URL: *http://www.cs.uni.edu/ wallingf/patterns/recursion.html*

Wallingford, E. (2001), 'The elementary patterns home page'.
URL: *http://www.cs.uni.edu/∼wallingf/ patterns/elementary/*

Watts, G. S. (1982), *The Revolution of Ideas: Philosophy, Religion & some Ultimate Questions*, Hale & Ironmonger, Sydney.

Webb, N. M. (1984), 'Microcomputer learning in small groups: Cognitive requirements and group processes', *Journal of Educational Psychology* **76**(6), 1076–1088.

Weber, K. (n.d.), 'Students' difficulties with proofs'.
URL: *http://www.maa.org/t_an_l/sampler/rs_.html*

Weber-Wulff, D. (2000), Combating the code warrior: a different sort of programming instruction, *in* 'Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education', ACM Press, pp. 85–88.

Weisberg, J. (n.d.), 'Being all that we can be', *Journal of Consciousness Studies* **10**(11), 33–38.

Werner, H. (1957), The concept of development from a comparative and organismic point of view, *in* D. B. Harris, ed., 'The concept of development', University of Minneapolis Press, Minneapolis.

West, D. (1997), 'Hermeneutic computer science', *Communications of the ACM* **40**(4), 115–116.

Weyl, H. (1959), *Philosophy of Mathematics and Natural Science*, Princeton University Press, Princeton.

White, H. (1987), *The Content of the Form: Narrative Discourse and Historical Representation.*, Johns Hopkins University Press, Baltimore, Maryland.

Whitehead, A. N. (1929), *Process and Reality: An Essay in Cosmology*, Cambridge University Press, Cambridge.

Whitehead, A. N. (1964*a*), *The Concept of Nature*, Cambridge University Press, Cambridge.

Whitehead, A. N. (1964*b*), Mathematics as an element in the history of thought, *in* R. W. Marks, ed., 'The Growth of Mathematics: From Counting to Calculus', Bantam Books, Inc., new York.

Wick, M. R. (2001), Kaleidoscope: using design patterns in CS1, *in* 'Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education', ACM Press, pp. 258–262.

Wiedenbeck, S. (1991), 'The initial stage of comprehension', *International Journal of Man-Machine Studies* **35**, 517–540.

WikiWikiWeb (2005), 'Front page'.
URL: *http://www.c2.com/cgi/wiki*

Williams, L. & Kessler, R. (2001), 'Experimenting with Industry's "Pair-Programming" Model in the Computer Science Classroom'.
URL: *http://www.pairprogramming.com/csed.pdf*

Winograd, T. (1983), *Language as a Cognitive Process*, Addison-Wesley, Reading, Massachusetts.

Winograd, T. (1996), Introduction, *in* T. Winograd, ed., 'Bringing Design to Software', ACM Press, New York.

Winograd, T. & Flores, F. (1987), *Understanding Computers and Cognition: A New Foundation for Design*, Addison-Wesley, Reading, Massachusetts.

Winslow, L. E. (1996), 'Programming pedagogy - a psychological overview', *ACM SIGCSE Bulletin* **28**(3), 17–22.

Wittgenstein, L. (1955), *Tractatus Logico-Philosophicus*, Routledge & Kegan Paul, London, England.

Woodworth, R. S. (1954), *Experimental Psychology*, third edn, Methuen & Co., London.

Young, P. (1996), 'Program comprehension'.
URL: *http://vrg.dur.ac.uk/misc/PeterYoung/pages/work/documents/lit-survey/prog-comp/*

Yourdon, E. (1975), *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs, New Jersey.

Yukawa, H. (1973), *Creativity and Intuition*, Kodansha International Ltd., Tokyo.

Zeilik, M. (n.d.), 'Classroom Assessment Techniques: Concept Mapping'.
URL: *http://www.flaguide.org/cat/minutepapers/conmap5.php*

Zimmer, J. A. (1998), 'Be ready to use recursion where warranted'.
URL: *http://www.mapfree.com/sbf/tips/recurs.html*

Zmud, R. W. (1979), 'Individual Differences and MIS Success: A Review of the Empirical Literature', *Management Science* **29**(10), 966–979.