# LDPL: A Language Designer's Pattern Language

by

Tiffany Winn, *B.Sc.(Comp.Sc)(Hons)*

School of Informatics and Engineering,
Faculty of Science and Engineering

November 22, 2006

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

Adelaide, South Australia, 2006

# Contents

# List of Figures

# List of Tables

# Abstract

Patterns provide solutions to recurring design problems in a variety of domains, including that of software design. The best patterns are generative: they show how to build the solution they propose, rather than just explaining it. A collection of patterns that work together to generate a complex system is called a pattern language. Pattern languages have been written for domains as diverse as architecture and computer science, but the process of developing pattern languages is not well understood.

This thesis focuses on defining both the structure of pattern languages and the processes by which they are built. The theoretical foundation of the work is existing theory on symmetry breaking. The form of the work is itself a pattern language: a Language Designer's Pattern Language (LDPL). LDPL itself articulates the structure of pattern languages and the key processes by which they form and evolve, and thus guides the building of a properly structured pattern language. LDPL uses multidisciplinary examples to validate the claims made, and an existing software pattern language is analyzed using the material developed.

A key assumption of this thesis is that a pattern language is a structural entity; a pattern is not just a transformation on system structure, but also the resultant structural configuration. Another key assumption is that it is valid to treat a pattern language itself as a complex, designed system, and therefore valid to develop a pattern language for building pattern languages.

One way of developing a pattern language for building pattern languages would be to search for underlying commonality across a variety of existing, well known pattern languages. Such underlying commonality would form the basis for patterns in LDPL. This project has not directly followed this approach, simply because very few pattern languages that are genuinely structural have currently been explicitly documented. Instead, given that pattern languages articulate structure and behavior of complex systems, this research has investigated existing complex systems theory – in particular, symmetry-breaking – and used that theory to underpin the pattern language. The patterns in the language are validated by examples of those patterns within two well known pattern languages, and within several existing systems whose pattern languages have not necessarily been explicitly documented as such, but the existence of which is assumed in the analysis.

In addition to developing LDPL, this project has used LDPL to critique an existing software pattern language, and to show how that software pattern language could potentially have been generated using LDPL. Existing relationships between patterns in the software language have been analyzed and, in some cases, changes to patterns and their interconnections have been proposed as a way of improving the language.

This project makes a number of key contributions to pattern language research. It provides a basis for semantic analysis of pattern languages and demonstrates the validity of using a pattern language to articulate the structure of pattern languages and the processes by which they are built. The project uses symmetry-breaking theory to analyze pattern languages and applies that theory to the development of a language. The resulting language, LDPL, provides language developers with a tool they can use to help build pattern languages.

# 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

Tiffany Winn

# Acknowledgements

I would first like to thank my supervisor, Paul Calder, for his support in so many ways. Thanks, Paul, for giving me the freedom to choose a research area that particularly interested me, providing high quality research feedback, being so loyal to me and all your research students, and supporting the life decisions I've made even when it's made thesis progress slower.

Experienced patterns guru James Coplien made several trips to Australia to work with the patterns group here at Flinders. During these visits he made a considerable contribution to the thesis.

Many of my fellow postgrads have been supportive in different ways. To Ron Porter, you are a wonderful person and a dear friend. Thanks so much for being a listening ear and offering good advice so many times. To Denise de Vries, thanks for all your help with Word. To Denise, Aaron Ceglar, Darius Pfitzner, and others in the KDIS group, thanks for your research feedback.

Staff members at Flinders have also offered support. Thanks to John Roddick for your contributions to the KDIS group, and Robert Goodwin for lots of little pieces of good advice, not to mention eggs and oranges! Thanks to Murk Bottema for reading a draft of the thesis. Thanks to Mike Schwarz for feedback on many of the biological examples. Thanks to the technical staff members, especially Rino Calacay and Michael Young, for your patience and willingess to answer even basic technical questions. Thanks to Leonie and the Postgraduate Students' Association (PGSA) for excellent and much needed advice and advocacy on all kinds of issues.

So many friends have given me support over the years. To Wendy and Simon, how do I say thanks for all the support? You've been wonderful. To Jen, Cat, and Anna, thanks so much for your friendship and support when I've really needed it. To Marianne V., it's been great to catch up when we can; thanks for your friendship. Thanks to Judy and Andrew for your regular commitment to the kids; it has made life so much easier for me. Thanks to the Prestons and the Manns for child care, and to Lisa especially for your friendship and support. To those at Walkerville Uniting Church who have been truly supportive – thanks so much for your respect, understanding, and graciousness. May you be recognized one day as the true leaders that you are! To Emma and the Monday Night Church community at Rosefield, thanks for loving and supporting me on my journey.

To my godkids, Jesse and Jordan, you are great fun as well as hard work. It is a privilege to share in your growing up.

Above all, I would like to thank my parents for supporting me in the life choices I have made, and especially for supporting me, Jesse and Jordan over the last two years. It has certainly been an adventure!

Tiffany Winn
November 2006
Adelaide

# Publications

The following publications have resulted from research done as part of preparing this thesis:

1. Winn, T. and Calder, P. (2002), Is this a Pattern? *IEEE Software* 19(1): 59-66, January/February 2002.
2. Winn, T. and Calder, P. (2002) A Pattern Language for Pattern Structure. In Proceedings of the Second Asian Pacific Conference on Pattern Languages and Programs (KoalaPLoP 2001).
3. Winn, T. and Calder, P. (2002) A Pattern Language for Pattern Language Structure. In Proceedings of the Third Asian Pacific Conference on Pattern Languages and Programs (KoalaPLoP 2002), pp. 49-58.
4. Porter, R., Coplien, J. O. and Winn, T. (2005), Sequences as a Basis for Pattern Language Composition. *Science of Computer Programming* 56(1-2): 231-249.
5. Winn, T. and Calder, P. (2005), A Language Designer's Pattern Language. In *Pattern Languages of Program Design 5*, Addison-Wesley, 2006.

# Chapter 1

# Introduction

Software needs to be *stable* – to behave predictably – and yet, at the same time, needs to be able to *change* – to adapt in a dynamic environment. In the range of approaches to building stable, adaptable software, one approach that offers promise is using patterns and pattern languages as a means of understanding how complex systems evolve over time.

The term *pattern* is widely used in a variety of ways in both technical and non-technical contexts. Alexander describes a pattern as "both a process and a thing" (1979, p. 247). At this level, his definition is consistent with the way a pattern is defined in physical systems, as a transformation on system structure that breaks symmetry. In a physical system, the pattern is both a process – a transformation – and the resulting structural configuration resulting from that transformation. For example, the formation of a snowflake involves vapor coagulating into globules that arrange themselves in particular ways; both the process of arrangement and the eventual structural configuration of the snowflake make the pattern.

The term *pattern language* refers to a collection of patterns that work together to generate a system (Coplien, 1996, p. 17) and was first popularized by architect Christopher Alexander (1977). Many software pattern and pattern language writers draw on Alexander's insights in developing their languages.

In the development of software pattern languages, one question that has not been well explored is how patterns interact with each other, and what insights can be drawn in this respect from both Alexander and theoretical understandings of patterns in other domains. Alexander put considerable effort into articulating how architectural patterns work together, but did not develop his understanding into a formal theory. The domain of physical systems is particularly interesting both because patterns in this domain have been extensively analyzed as transformations on system structure that break symmetry, and because physical systems are both continually evolving and inherently stable: "Natural systems must be stable; that is, they must retain their form even if they're disturbed" (Stewart and Golubitsky, 1992). Software system designers who seek to build software systems that are stable and yet able to adapt ought therefore to be able to build better systems by learning from analysis of physical systems.

This thesis uses Alexander's (1977; 1979) work and theory about symmetry breaking and patterns, such as Stewart and Golubitsky's (1992) research focusing primarily on natural, physical systems and Coplien and Zhao's (2002; 2001) work on symmetry and software patterns, as a basis for further exploration of how patterns work together. It aims primarily to help writers of pattern languages write better pattern languages through better understanding what a pattern language is and how the patterns in a language work together. Better pattern languages will in turn help to generate better systems. The form of the work in this thesis is itself a pattern language that articulates the structure of pattern languages and key processes by which they are formed and evolve, and thus guides the building of a properly structured pattern language.

A strong component of the pattern ethos is that patterns are validated by positive examples, rather than underlying theory. So, for example, the "proof" that a pattern exists is that it can be demonstrated to be present in a number of systems, rather than, say, a mathematically based theoretical proof. For this reason, the language developed in this thesis is validated using multidisciplinary examples. Any work that attempts to infer underlying structure from existing, positive examples can only posit a best guess. Further, any work that attempts to deduce some notion of process from existing structure can only posit a plausible explanation. This thesis, therefore, does not claim absolute correctness, but rather offers a sound explanation of how best to build a pattern language.

In addition, a particular difficulty with validating the patterns in the language developed here by using existing examples is that in many domains, including software, there are very few existing generative pattern languages. Generativity is a key to Alexander's understanding of pattern, and is also reflected in symmetry-breaking explanations. Generativity means that patterns work together with other patterns to generate structure. A pattern language is therefore not simply a guide to good design, but articulates in some sense the structure of a well-structured system. In a similar sense to that with which a dressmaking pattern gives a picture of what will be created with it, a software pattern gives a picture of the software structures it builds. Since there are few existing, generative languages, it is necessary for this work to rest more on underlying theory than would be desirable in an ideal situation. As more generative languages are developed, the validity of this work will be better able to be tested.

The remainder of this chapter is structured as follows: Section 1.1 outlines overall goals of the research, Section 1.2 lists contributions of the thesis, Section 1.3 provides a historical perspective on patterns and pattern languages in the software domain, and Section 1.4 outlines the structure of the rest of the thesis.

## 1.1 Research Goals

The main research goals of this thesis are as follows:

- To better define the pattern concept.
- To better define the way patterns work together in a pattern language.

Defining patterns and pattern languages is difficult, because they do not lend themselves to prescriptive, formal definitions, and are not bound by such definitions. Yet understanding patterns and pattern languages at a theoretical as well as practical level is critical. Pattern writers need to understand patterns and pattern languages at both levels if they are to identify them, use them well, and distinguish them from similar-seeming non-patterns. The lack of clarity around the pattern concept, including the lack of clarity on how patterns work together, must be addressed if the concept is to retain its force (Winn and Calder, 2002).

This research does not aim to develop a formal theory about how patterns work together, or to formalize patterns in any way. Instead, it seeks to use informal analysis of a variety of patterns and pattern languages as a basis for establishing a theoretical foundation for patterns and pattern languages that better articulates how patterns work together.

## 1.2 Contributions

The key contributions of this thesis are as follows:

- ***It establishes foundations for a generative semantic model for pattern languages.*** Existing analyses of pattern languages, at least within a software context, tend to focus on syntax, or the mechanics of writing individual patterns, rather than semantics, or how patterns work together to generate a system. Analyses that do focus on semantics tend to be limited to object-oriented design patterns. This work builds on Alexander's (1979) understanding of how patterns in a language work together and on other relevant work, such as theory on patterns and symmetry breaking (Stewart and Golubitsky, 1992), to more explicitly articulate how the patterns in a language work together to generate a system.
- ***It articulates the structure of pattern languages, and the processes by which they are built.*** Many pattern languages in existence today are little more than collections of patterns that relate to each other in some way. Exactly how the patterns relate to each other is often not well defined, and what whole, or larger structure, the patterns combine to generate is often unclear. In other words, the structural relationships between patterns are often not well defined, and the processes by which the language is generated tend also not to be well defined. In addition, few researchers who discuss pattern language structure also discuss the kind of processes that enable that structure to be generated. Alexander (1979) is a notable exception; he is clear that smaller patterns differentiate larger ones (1979, pp. 305-324, 365-384), and that only certain kinds of processes make possible the generation of good structure (Alexander, 2002b, pp. 203-228). This work builds on Alexander's and other relevant work to more explicitly articulate the structure and process of pattern languages.
- ***It uses a pattern language to express the structure of pattern languages, and the processes by which they are built.*** The structure of pattern languages and the processes by which they are built can be potentially articulated in a variety of ways. This thesis articulates those structures and processes in the form of a pattern language. A key advantage of this approach is that it combines theory about patterns with their use. Rather than describing a theory that is separate from but used in the application of patterns, the theory is itself explicitly part of the

application process. Combining theory and practice is a key feature of a patterns-based design approach (Winn and Calder, 2002).

- ***It presents a pattern language that can be used to help build pattern languages, and to analyze existing pattern languages.*** The pattern language developed in this thesis, called the Language Designer's Pattern Language (LDPL), articulates how to build a pattern language, and what kind of structure a pattern language ought to have. LPDL can be used to help pattern language developers to build better languages, and provides a framework for analyzing existing languages. The thesis describes an analysis of the C++ idioms language (Coplien, 2000a) using LDPL. The patterns in the idioms language and the existing relationships between them are analyzed and changes to the idioms language suggested based on this analysis.
- ***It uses symmetry-breaking theory to analyze existing pattern languages and systems.*** As part of its development, this thesis analyzes relationships between patterns in several different pattern languages and systems, in terms of symmetry breaking. Only a few patterns, particularly software patterns, have previously been analyzed in terms of symmetry breaking (Coplien and Zhao, 2001; Coplien, 1998b). Given the importance of symmetry breaking to analysis of complex systems in a variety of domains (Stewart and Golubitsky, 1992; Alexander, 2002a), symmetry breaking analysis of patterns is an important area of research. In addition, the language developed in the thesis can potentially be used by developers of pattern languages and shows how to apply symmetry-breaking theory in that development.
- ***It defines a precise relationship between patterns of different levels of scale.*** Many writers of pattern languages use pattern language diagrams to describe the connections between patterns in the language, but the nature of those connections are rarely explicitly defined. The language presented in this thesis defines those connections.
- ***It draws together theory on the nature of order, symmetry breaking, and pattern languages.*** Many different researchers have investigated patterns and their role in determining how systems evolve over time. The thesis pulls together research focusing on system structure, patterns

as symmetry breaking, and pattern languages, so as to better articulate the role of pattern languages in defining the behavior of complex systems.

- ***It identifies common threads in discussions on patterns from a wide variety of domains and uses these to better define the pattern concept.*** The term "pattern" is widely used in a significant number of domains, both colloquially and in scientific and technical contexts. This thesis identifies common features across definitions in very different domains, developing a concept of pattern that is both broad enough to cut across different domains and concise enough to be of value.

## 1.3 History of Research on Patterns in Software

Although he does not explicitly mention patterns, Peter Naur was perhaps the first person to recognize the importance for software designers of architect Christopher Alexander's work. In his report on the 1968 NATO Conference in Garmisch, Germany (Naur and Randell, 1969, pp. 35, 45), Naur noted the similarity between software design, architecture design and civil engineering, particularly where the architectural and civil engineering work involves large heterogeneous constructions, such as towns. He cited Alexander's (1964) early work, as a potential source of design wisdom for software engineers.

Kent Beck and Ward Cunningham were amongst the first software developers to explicitly work with patterns (Cunningham, 2003). In early 1987, they were consulting with a team struggling to specify the design of a user interface software system. They were familiar with Alexander's (1979) work on patterns in architecture; in particular, they were mindful of Alexander's advice that a pattern language could help bridge communication gaps between users and developers, and that the occupiers of a building ought to be involved in its design. Cunningham devised a small pattern language guiding the development of a user interface system in Smalltalk and invited user representatives to design the required system based on that pattern language. He and Beck were amazed by the speed and elegance with which the design was achieved with the aid of Cunningham's patterns.

Erich Gamma's (1993) initial work had no foundation in Alexander, but Gamma was one of the first software developers to recognize that recurring design structures are important. During development of ET++ (Weinand *et al.*, 1988), a financial engineering tool and comprehensive C++ library, Gamma noticed that some design structures emerged repeatedly (Eggenschwiler and Gamma, 1992). He argued that use of these structures typically improved a design's flexibility, reusability and elegance.

In 1992, James Coplien published *Advanced C++ Programming Styles and Idioms* (1992), noting that programming idioms are "reusable 'expressions' of programming semantics, in the same sense that classes are reusable units of design and code" (1992, p. xi). C++ seeks to provide flexibility for programmers by providing a set of building blocks from which they can create not only programs, but their own building blocks – their own idioms. Even though these standard programming techniques or idioms are not actually part of the language itself, they are fundamental building blocks for C++ programmers:

> … nothing in the C++ language itself establishes `while (*cp1++ == *cp2++);` as a fundamental building block, but a programmer unfamiliar with this construct would not be perceived as a fluent C++ programmer …
>
> (Coplien, 1992, p. vi)

Learning to program well in C++ requires understanding not only the language syntax, but also what can be done with that syntax; it requires understanding idioms. Coplien's C++ idioms are the product of years of his and others' experiences in programming in C++ and represent, in some sense, C++ patterns. They form the basis for Coplien's (2000a) C++ idioms language, which is used extensively as a source of examples in Chapter 5 and further analyzed in Chapter 6.

Many people working independently in areas related to patterns had the opportunity to explore common threads at several Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) workshops in the early 1990's, particularly the *Towards An Architecture Handbook* workshops led by Bruce Anderson (Coplien, 1996, p.45-48). Through these workshops, many software developers were both introduced to Alexander's architectural patterns and began to take an interest in patterns. Many of those who attended these workshops went on to become key researchers in the patterns community.

Peter Coad (1992) was one of the earliest in the software patterns community to recognize the interdisciplinary nature of patterns. He cited examples of patterns in a variety of disciplines, including music, psychology, literature and numismatics. Coad's primary focus, though, was on object-oriented patterns, and he analyzed Alexander's (1979) work with a view to incorporating relevant insights into his own. Coad noted Alexander's emphasis on patterns as geometric structures that act as fundamental building blocks in a domain, and sought to develop object-oriented patterns that represented such fundamental building blocks. Alexander's insight that the relationships between entities are key to identifying patterns led Coad to argue that it is the relationships between classes and objects, rather than the classes and objects themselves, that define object-oriented patterns. His own patterns are centered on relationships; Broadcast, for example, captures a particular communication protocol between objects. More recently, and in contrast to his earlier work, Coad (1995) has argued that a pattern is simply a template - a plan for a solution - and not, as Alexander argues, an actual, concrete solution to a problem in a context (Alexander, 1979, p.247). Coad's templates/patterns still represent commonly used object relationships, but lack contextual information, as discussed in Section 3.1.1.

Doug Lea brought a background in object-oriented design to discussion of patterns, and by early 1994 had published an introduction to Alexander's work for object-oriented designers (Lea, 1994). The introduction puts the use of patterns into the context of Alexander's overall views about design and the ultimate purpose of design: to improve the human condition. Alexander emphasizes the importance of not slavishly adhering to rigid, formal design methods, but of having design methods that are flexible and able to adapt. Lea argues that patterns are central to such a process because they exhibit certain properties: encapsulation, generativity, equilibrium, abstraction, openness, and composability. Encapsulation refers to knowing when and when not to apply the pattern, generativity refers to the creative power of patterns, equilibrium to the balancing of forces, abstraction to the commonality across different solutions captured by a pattern, openness to the capacity to extend patterns to arbitrarily fine levels of detail, and composability to the way patterns work together with finer-grained patterns elaborating more coarsely-grained patterns. Lea argues that in an object-oriented context, "[patterns] may be viewed as extending the definitional features of classes" and that the best classes exhibit the above-listed properties. Lea further argues that object-oriented concepts may offer insights to strengthen pattern-based approaches to design. For example, extensions to object-oriented modeling and design might be of help in representing patterns.

Frank Buschmann *et al.* (1995; 1996) are some of only a few authors to explicitly recognize the importance of scale in the organization of patterns. They describe what they term a pattern *system* for software architecture that encompasses patterns on three distinct levels of scale: architecture, design, and idiom levels. Connections across levels of scale are highlighted as part of individual pattern descriptions. Buschmann's pattern system is discussed further in Section 2.3.1.

In 1995, Wolfgang Pree published the book *Design Patterns for Object-Oriented Software Development* (1995). In this book, Pree documents what he calls meta-patterns: recurring structures within existing design patterns. Most people in the software patterns community, however, would not call Pree's meta-patterns design patterns, and some would question whether they are actually patterns at all (Coplien, 1996, p.28).

Perhaps the most important outcome of the OOPSLA workshops was the formation of the Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Together, they wrote a book (Gamma *et al.*, 1995) consisting of a catalog of object-oriented design patterns: recurring solution structures used in object-oriented development. The book was hugely successful, being largely responsible for bringing patterns into mainstream software development; the number of people writing, using, and writing about patterns began to grow rapidly.

## 1.4 Structure of the Document

The remainder of the document is structured as follows. Chapter 2 provides an introduction to patterns and pattern languages. Chapter 3 puts the work of this thesis in context by outlining related work in the areas of design, especially software design, and pattern language theory. Chapter 4 overviews the symmetry breaking theory that is used in the development of the language designer's pattern language described in Chapter 5. In Chapter 6, an existing software pattern language is analyzed using the work presented in Chapter 5. Finally, Chapter 7 presents conclusions and outlines areas for future research.

# Chapter 2

# Foundations of Patterns

This chapter provides an introduction to patterns and pattern languages. Section 2.1 provides a basic understanding of what a pattern is, and Section 2.2 follows this up with detailed examples of patterns from several domains. Sections 2.3 and 2.4 similarly focus on pattern languages. Section 2.3 covers basic concepts and Section 2.4 explores these concepts in the context of detailed examples. Following on from the general overview provided by the earlier sections, Section 2.5 defines four key elements of a pattern – context, forces, problem, and solution – and provides a brief overview of how those key elements shape the place of a pattern within a language. As a whole, the chapter provides the background on patterns and pattern languages necessary to understanding the pattern language that forms a major part of this thesis.

## 2.1 Pattern Basics

The term "pattern" is widely used in many domains. The term is used colloquially, as well as in scientific and technical contexts, in a variety of different ways. For example, a pattern might be defined in an artistic context as a "regularly repeated arrangement, especially a design made from repeated lines, shapes or colors on a surface" (Cambridge University Press, 2003). In the context of understanding how a business operates, a pattern might be "a particular way in which something is done, organized or happens" (Cambridge University Press, 2003), or where a physical artifact is being built, "a model or original used as an archetype" (Pickett, 2000). In a software system, a pattern could be a particular recurring structural relationship between classes or object instances (Coad, 1992). Patterns in architecture describe fundamental structural characteristics of buildings and towns (Alexander *et al.*, 1977).

In a physical system, the term pattern is used to refer to the configuration that forms as a result of a transformation that breaks symmetry (Stewart and Golubitsky, 1992). While there are some common features across definitions, there is no definitive, universal definition of the pattern concept, and the extent of the connection between the use of the term pattern in one domain, such as software, with its use in another domain, such as architecture, remains unclear.

Within the software domain, the term "design pattern" was used by Gamma *et al.* (1995) to describe a collection of recurring problem/solution pairs for object-oriented design that they catalogued. Their collection has come to be known as the Gang of Four (GoF) patterns catalog. They argue that a design pattern "names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design" (Gamma *et al.*, 1995, p.3). The GoF design patterns describe both key design problems for object-oriented developers, and strategies commonly used by experienced developers to address those problems. Those strategies encompass both static and dynamic class and object hierarchies and other structures.

In the software design literature, the term "design pattern" tends to be used to refer to the GoF patterns and other similar collections of patterns, where the context is object-oriented and the focus is on individual patterns solving isolated design problems, rather than how the patterns work together. The terms "pattern" and "software pattern" tend to be used more generally, and may indicate an approach to patterns that is more focused on patterns that work together as part of a pattern language to build a system. In this thesis, the more general use of the term is appropriate since the focus is on how patterns work together, and the context is much broader than recurring solution structures in object-oriented programs.

Given this context, this section seeks to provide a basic understanding of patterns that is accurate for the software context in which the current work is based, but also broad enough to encompass the inherently multidisciplinary nature of the pattern concept. Section 2.1.1 highlights pattern definitions offered by software developers. Many of these definitions draw on analogies with non-software disciplines, or present a general pattern definition and rely on a software example to provide context. Section 2.1.2 points out important insights on patterns from architect Christopher Alexander. Section 0 emphasizes the multidisciplinary nature of the pattern concept, briefly exploring links between patterns in software and other domains. Section 2.1.4 briefly outlines pattern theory ideas that are not limited to one particular domain.

### 2.1.1 What is a Pattern: Insights from Software Developers

The literature on patterns, perhaps consciously, provides no authoritative and complete definition of what constitutes a pattern. Instead, it offers a collection of definitions from different perspectives and contexts. Together, these definitions offer a beginning point for better understanding the pattern concept.

A software pattern is perhaps most frequently described as a *solution to a problem in a context* (Schmidt *et al.*, 1996; Coplien, 1996, p.2; Gamma *et al.*, 1993, p.3). Within a given domain, what may appear to be very different problems often turns out to be the same basic problem occurring in different contexts. A software pattern identifies such a recurring problem and a solution, describing them in a particular context to help developers understand how to create an appropriate solution. Patterns thus capture and explicitly state general problem-solving knowledge that is usually implicit and gained only through experience:

> When experts work on a particular problem, it is unusual for them to tackle it by inventing a new solution that is completely distinct from existing ones. They often recall a similar problem they have already solved, and reuse the essence of its solution to solve the new problem. This kind of "expert behavior", the thinking in problem-solution pairs, is common to many different domains …
>
> (Buschmann *et al.*, 1996, p.2)

Explicitly stating key design knowledge in a pattern can bring that knowledge to the attention of developers who would otherwise be unaware of it. Developers can use that knowledge to solve what appears to be a new problem with a tried-and-true solution, thus improving the design of new software. But a software pattern is more than just a specification for a solution:

> I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.
>
> (Coplien, 1996, p. 3)

A pattern provides some sort of picture of the geometry or shape of the potential artifacts it describes. For software, this means understanding at a big picture level how the software works and at a design level the relationships that the software attempts to capture (Winn and Calder, 2002). A pattern is also both process and product; it articulates both the process required to generate an artifact, and the artifact that will be generated. A pattern therefore does more than just describe the characteristics of a good system. It also teaches how to build such systems, and for this reason is often described as *generative* and *structural*:

> … patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context.
>
> (Coplien and Schmidt, 1995, p. xi)

Furthermore, the structure that a pattern builds *balances forces*. Understanding a problem requires understanding both the forces that bring it about and how those forces interact. Developing a solution to a problem therefore requires balancing possibly conflicting forces in such a way that the solution structures developed are *stable* enough to be effective. The fact that a pattern encompasses problem, solution and context enables it to articulate the centrality of forces to problem solving, and to provide stable solutions to problems:

> The documentation of a pattern goes beyond documenting a problem and its solution. It also describes the *forces* or design constraints that give rise to the proposed solution. These are the undocumented and generally misunderstood features of a design. Forces can be thought of as pushing or pulling the design towards different solutions. A good pattern balances the forces.
>
> (Weiss, 2003)

Viljamaa provides a simple example of a situation where forces need to be balanced:

| | |
|---|---|
| *Programmer1:* | We must do it my way because otherwise it is too slow. |
| *Programmer2:* | But that would mean we need lots more memory. |
| *Programmer3:* | Both of your solutions take too long to code. |
| *The Patternist:* | Hey, I found a solution in my patterns handbook that seems to reasonably balance your desires. |

(Viljamaa, 1995)

### 2.1.2 What is a Pattern: Insights from Alexander

Christopher Alexander's foundational work on patterns and pattern languages in architecture has inspired many in the software community. It is at least partly due to Alexander's influence that a software pattern is commonly described as a solution to a problem in a context:

> Each *pattern* is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
>
> (Alexander, 1979, p.247)

Alexander also emphasizes both the importance of forces to patterns and their generativity:

> As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
>
> …
>
> The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive and a description of the process which will generate that thing.
>
> (Alexander, 1979, p.247)

Alexander regards patterns as fundamental to architecture:

> Once we understand buildings in terms of their patterns, we have a way of looking at them which makes all buildings, all parts of a town similar … We have a way of understanding the generative processes which give rise to these patterns.
>
> (Alexander, 1979, p.11)

He notes, however, that although patterns "… seem like elementary building blocks, [they] keep varying and are different every time they occur" (Alexander, 1979, p.84). Patterns articulate recurring structure, but that structure changes according to context. Alexander argues that the invariant structure that a pattern articulates has to do with relationship, rather than being embodied in what would traditionally be seen as an entity. It is the structural relationships that remain constant across a variety of contexts. For example, while each gothic cathedral has a nave and aisles, the particulars of nave and aisle are quite different from one cathedral to another. Neither nave nor aisle by itself forms a pattern. The pattern is the structure generated by the invariant relationship between nave and aisle in gothic cathedrals: "within a gothic cathedral … the nave is flanked on both sides by parallel aisles" (Alexander, 1979, p.90). Alexander thus describes a pattern as follows:

> [A pattern is] a morphological law, which establishes a set of relationships in space.
>
> This morphological law can always be expressed in the same general form: X -> r (A, B, …), which means:
>
> Within a context of type X, the parts A, B, … are related by the relationship r.
>
> (Alexander, 1979, p.90)

### 2.1.3 The Multidisciplinary Nature of the Pattern Concept

While the way the term pattern is used in one domain might be at least superficially different from the way it is used in another domain, the existence of entities called patterns is recognized in many different domains. This section seeks to explore, briefly and at an informal level, any underlying universality in the pattern concept. To this end, it briefly reviews patterns in different domains, emphasizing recurring themes, and provides an overview of existing formal definitions of patterns that provide some basis for a multidisciplinary understanding. Common threads emerging from the patterns highlighted in this section include that patterns are generative, structural, and identify recurring themes characterizing complex systems.

Coad (1992) provides a useful starting point for investigating the multidisciplinary nature of patterns. He highlights the multidisciplinary nature of the pattern concept by providing brief comparisons between patterns in a number of domains. For example, he describes a psychological pattern as "a thinking mechanism that is basic to the brain's operation, helping one to perceive things quickly". He sees a chess pattern as a set of moves and their place in an overall strategy, a linguistic pattern as the way that smaller language units are combined to form larger units, and an aviation pattern as a collection of landing approaches, turns and altitudes for an aircraft coming in to land. Coad concludes that a common feature across all these domains is that patterns deal with more than the smallest, atomic elements in a domain, and that they amount to repeating patterns of relationship among elements. For this reason, Coad defines an object-oriented pattern as a recurring relationship between objects and classes.

Russian scientist Genrikh Altshuller (1984) studied the creativity of the inventor by analyzing hundreds of thousands of invention descriptions from world-wide patent databases. He found that there are key, recurring strategies that facilitate the solution of complex, technical problems in a wide variety of domains. He went on to argue that all technological systems evolve according to objective and predictable patterns, regardless of their particular domain. For example, one of those patterns concerns solving problems that require elimination of a technical contradiction. Rather than a typical trial-and-error approach, Altshuller proposes an approach whereby a new substance is introduced into the system, together with an appropriate "field". The new substance, when acted on by the field, changes the forces present in the system in such a way that they are balanced. He argues that this approach can and has been successfully used in numerous diverse situations. He gives as an example a factory that needs to test several hundred different types of soil. Building several hundred different testing grounds could be expensive and complex; experimental accuracy must be traded off against cost and complexity of construction. Using a trial-and-error approach, a typical solution will not solve the technical contradiction, but will compromise both opposing forces. Using Altshuller's approach of looking for a new substance and an appropriate field, a solution that resolves the contradiction can easily be found. A ferromagnetic powder can be added to the existing soil, and a magnetic field can be applied to the mixture of soil and ferromagnetic powder. Activating the magnetic field can change the characteristics of the mix as needed so that the best solution to the problem can be found.

Bergin (2002) documents a number of pedagogical patterns based on his experience of teaching many university-level computer science courses. These patterns aim to capture expert knowledge in the practice of teaching and learning. They address recurring problems in the teaching domain, such as motivating students, evaluating students and selecting and sequencing materials. Bergin emphasizes the importance of patterns to sharing knowledge usually only gained through experience. While most of the patterns he documents represent common knowledge for skilled instructors, documenting the patterns allows that knowledge to be more easily shared:

> The intent is to capture the essence of the practice in a compact form that can be easily communicated to those who need the knowledge. Presenting this information in a coherent and accessible form can mean the difference between every new instructor needing to relearn what is known by senior faculty and easy transference of knowledge of teaching within the community.
>
> (Bergin, 2002)

Coplien (1999) has investigated patterns that contribute toward building the dynamic structure of a T-ball swing. Each of the patterns is structural, showing how to position one's body and how the structures of the body should move to create the dynamic structure of a swing. The collection of T-ball patterns as a whole emphasizes the way that patterns work together to build a system.

Senge (1990) argues that complex systems are characterized by recurring patterns and that understanding the behavior of complex systems requires recognizing those fundamental patterns. One of those patterns is that cause and effect are *not* close in time and space, contrary to the assumptions that most people make:

> … a fundamental characteristic of complex humans systems [is that] "cause" and "effect" are not close in time and space. By "effects," I mean the obvious symptoms that indicate that there are problems – drug abuse, unemployment, starving children, falling orders, and sagging profits. By "cause" I mean the interaction of the underlying system that is most responsible for generating the symptoms, and which, if recognized, could lead to changes producing lasting improvement. Why is this a problem? Because most of us assume they *are* – most of us assume, most of the time, that cause and effect *are* close in time and space.
>
> (Senge, 1990, p. 63)

The patterns that Senge identifies contribute to generating the behavior of complex systems. Senge emphasizes that patterns are about interrelationships rather than linear cause-effect chains, and are about a process of change rather than an instant in time.

His approach builds on recognizing recurring structures that characterize deeper patterns underlying events and details.

Bern (1964) describes psychological games played by people in various social situations. Procedures, rituals, pastimes and games are all ways of structuring time, and can be analyzed in terms of social transactions. Each participant in a social transaction can be driven by a Parent, Adult, or Child ego state. Procedures, rituals and pastimes are relatively simple, and can be analyzed using simple transaction series. Games are more complex. A game can be defined as "an ongoing series of complementary, ulterior transactions progressing to a well-defined, predictable outcome" (Bern, 1964, p. 44). Key characteristics of games are their ulterior quality and payoff. People playing games are not open and honest in their motives; every game is basically dishonest. The hidden agenda driving the game produces some kind of payoff that holds the key to the game player's true motive. The key function of games is to be a barrier to intimacy by creating a situation in which honest, open interaction is avoided.

Games as described by Bern are structural in the sense that they describe key, recurring relationships between entities. The entities are human beings and analysis of the games people play consists of describing a person's structure in terms of Parent, Adult, and Child ego states, and describing interactions between people involving different states. There are strong similarities between Bern's concept of a game and the concept of pattern as it is used in this thesis. Bern notes, however, that most of the games do not, in his view, contribute to building healthy relationships, because they are fundamentally dishonest and thus prevent true intimacy. He only mentions a few patterns that are "good patterns". While the style of presentation Bern uses is reminiscent of Alexander's (1977) presentation of architectural patterns, with each game being described using Thesis, Antithesis, Aim, Roles, Dynamics, and Examples, and other sections, the kind of patterns Alexander focuses on are different; Alexander focuses on documenting "good" patterns.

In summary, a pattern articulates fundamental, recurring system structure. As such, it embodies key design knowledge that can be used in building and maintaining systems. It provides a solution to a problem that balances key forces in a given context. The best patterns are generative, teaching how to build the solution they propose, rather than just explaining it.

### 2.1.4 Multidisciplinary Pattern Theory

Few theories about patterns and how they work together are explicitly grounded in any kind of formal mathematics. Two notable exceptions that contribute to developing a multidisciplinary understanding of patterns are outlined in this section. The first relates to symmetry and symmetry breaking; in physics, patterns are well recognized as structural transformations that break symmetry. The second, Grenander's General Pattern Theory, is more focused on pattern recognition than on the generative processes that give rise to patterns. It is, however, extensive and well recognized in many fields. It is also grounded in the mathematics of group and set theory, and concepts of symmetry.

In physics, patterns have long been recognized as the results of a phenomenon called symmetry-breaking (Stewart and Golubitsky, 1992). Intuitively, symmetry relates to a correspondence in size, shape and relative position around some center or axis. Symmetry breaking concerns how symmetry is redistributed in a system when the system needs to change. As Stewart and Golubitsky note, scientists and mathematicians have begun to realize that symmetry breaking is significant to the formation of patterns:

> This paradox, that symmetry can get lost between cause and effect, is called *symmetry-breaking*. In recent years scientists and mathematicians have begun to realize that it plays a major role in the formation of patterns.
>
> …
>
> From the smallest scales to the largest, many of nature's patterns are a result of broken symmetry; and our aim is to open your eyes to their mathematical unity.
>
> (Stewart and Golubitsky, 1992, p. xviii)

Coplien and Zhao (2001) argue that symmetry breaking is fundamental to patterns, and that attempts to formalize software patterns should be based on notions of symmetry. They are working on identifying and formalizing symmetries in software:

> … [a software pattern is characterized by] a structure [resulting] from breaking an original symmetry, where the symmetry is defined in terms of an invariant in a system.
>
> (Coplien and Zhao, 2001)

Symmetry breaking and patterns are discussed in more detail in Chapter 4.

Grenander (1996) has developed an extensive, mathematical theory of patterns that is powerful enough to represent knowledge about complex systems. He also recognizes the connection between symmetry and pattern: patterns are made up of parts that have particular relationships, and the symmetry group of a relationship defines the transformations under which key invariants in the pattern are preserved. Grenander's theory is designed to build algorithms that can recognize patterns, rather than to identify the generative processes that give rise to patterns. Those algorithms focus on how patterns work together, but use the after-the-fact structure of a composition of patterns in the recognition process, rather than focusing on how that structure has been generated. While Grenander does not consider the transformations that give rise to a pattern as part of the pattern, he is one of few to offer a sophisticated mathematical definition of a pattern:

> Given an image algebra *I* we shall understand by a pattern any *S*-invariant subset *P* of *I*, and by a pattern family a partition of *I* into patterns.
>
> (Grenander, 1996, p. 93)

An image algebra is a set of equivalence classes defined with respect to some equivalence relation. Any two equivalence classes in the set are disjoint and the union of all the equivalence classes forms the image algebra. The elements of an equivalence class are the same, or equivalent, with respect to a given equivalence relation. For example, if *I* consists of images of various cars, given the equivalence relation "having the same color", one equivalence class for *I* is the set all images of green cars.

*S* is a similarity group; that is, a set of transformations such as rotation, reflection, and translation. For *P* to be *S*-invariant means that *P* does not change when any transformation in *S* is applied to it. *P* is a pattern; it is a subset of images of *I*, and the transformations that create that subset are defined separately to the pattern.

In summary, the formation of patterns in physical and biological systems has been linked to symmetry breaking. The recognition of patterns in a variety of complex systems has also been linked to symmetry breaking.

## 2.2 Examples of Patterns

An important principle of software pattern communities worldwide is that patterns are best understood in the context of concrete examples, and the following sections each focus on examples from a particular domain. Section 2.2.1 details software examples. Given the importance of Alexander's (1977; 1979) architectural work to the development of software patterns, architectural patterns are the focus of Section 2.2.2. Section 2.2.3 focuses on biological patterns. The biological patterns are of particular interest because of the existence of formal, mathematically grounded analysis of such patterns compared with the lack of such formalization in other domains. Section 2.2.4

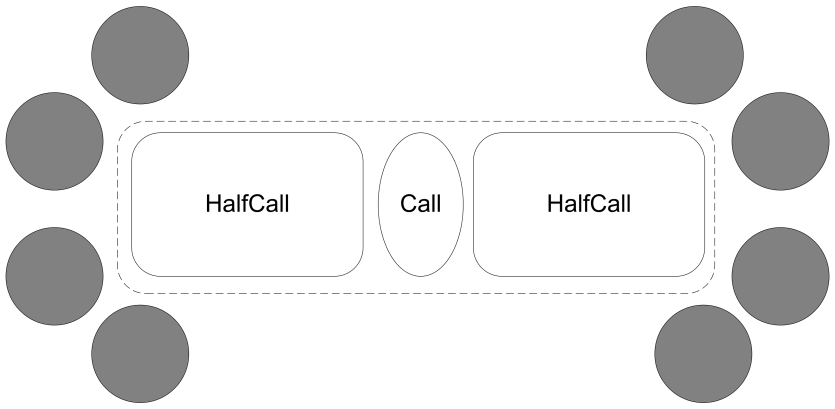

**Figure 2-1: A HOPP object, shown within the dotted lines, together with clients, shown as darker circles (based on (Coplien, 1998b)). The area labeled Call represents interactions between the two HalfCall objects.**

provides examples of patterns from several domains not covered in earlier sections, with a view to further demonstrating the breadth of the pattern concept. Building on common features of patterns noted in Section 2.1, all examples, regardless of domain, are analyzed in terms of the key, recurring knowledge that the pattern captures, the structure that the pattern helps generate, and the forces that the pattern resolves.

### 2.2.1 Software Patterns

One of the most universally recognized patterns is Half Object Plus Protocol (HOPP) (Meszaros, 1995), illustrated in Figure 2-1. HOPP is a pattern for distributed computer systems. Such systems must often be implemented across multiple address spaces.

They can sometimes be decomposed into objects that live in exactly one address space, but other times a concept exists in more than one space. For example, in designing a distributed call processing system the concept of a phone call, embodied in the capacity to establish and terminate a phone call, needs to exist in address spaces at both ends of the call.

Implementing a concept that exists across address spaces can be done in a number of ways. For example, all but the most basic processing could be done in one address space. Alternatively, a separate object and protocol could be designed for each address space, and then interaction across address spaces managed by another protocol.

HOPP suggests that where a concept needs to be implemented across two or more address spaces, it should be done such that there is a half-object in each address space and clients can interact with each half-object as if it were the original, whole object. This is achieved by splitting the original object into half-objects, but retaining those half-objects, at least conceptually, as part of a single, overarching object. The communication protocol between half-objects is also part of the one whole object. The original object is hence transformed into a HOPP object: half-objects plus protocol. The HOPP pattern further specifies that half-objects be designed such that each half-object is as like the original object as possible and the protocol carries only a minimum of information between address spaces.

HOPP is structural: it defines the structure of two objects by creating them as half-objects, and defines a particular structural relationship between them. The forces that HOPP resolves have to do with placing objects into address spaces such that issues of complexity, distribution, information availability, cost, and performance can be appropriately managed.

Another very well known software pattern is Handle/Body (Coplien, 1992; Vihavainen, 2001), which is used to separate interface from implementation in C++ programs, and is illustrated in Figure 2-2. Handle/Body splits a user-defined type into two separate classes. The first class, the handle, acts like a built-in type. It presents only the class interface to the user. The other class, called the body, defines the implementation. The two classes act together to define a single object; the handle part forwards operation calls to the body class.

Handle/Body allows changes to be made to implementation without forcing recompilation of client code. It also allows implementation code to be hidden from the client. Whilst Handle/Body carries a performance cost, due to the extra level of indirection, it makes supporting variable-sized objects much easier. The programmer can take full advantage of the polymorphism provided by inheritance when used for subtyping: the additional attributes of a derived type can be maintained when assigned to a *handle* for the base class, whereas they would be lost when assigned to an *instance* of the base class, due to lack of space. The solution is similar to using object references, but easier to work with. In the domain of C++ programming, Handle/Body is a structure that is a solution to a recurring problem.

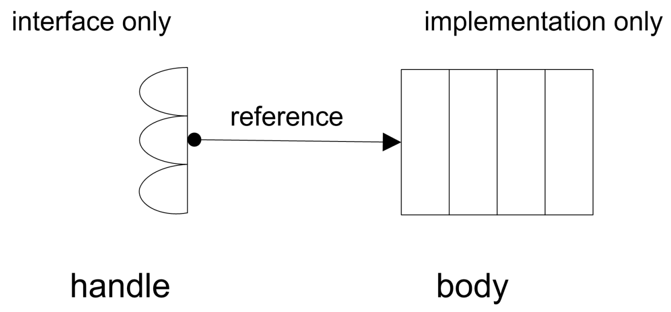

**Figure 2-2: The structure of a Handle/Body object. The Handle object presents the class interface and the Body object defines the implementation; the two objects are linked with a reference and together make the Handle/Body object.**

The structure that Handle/Body defines is that of handle and body objects and the relationship between them. The forces that it resolves concern memory management. Handle/Body provides the convenience of scope-based memory allocation for entities that may not actually be in program scope. Coplien (2000a) describes the Handle/Body idiom in pattern form as part of a pattern language for building an algebraic hierarchy in C++.

The Gang of Four (GoF) design patterns (Gamma *et al.*, 1995) have an important place in the history of software patterns. These patterns popularized the use of the term pattern in software and made the use of patterns a recognized part of mainstream software development. Two of the GoF patterns are highlighted here. Both are well known and extensively used by software designers.

The Mediator pattern (Gamma *et al.*, 1993, pp.273-303) defines an object that "encapsulates how a set of objects interact". When Mediator is used, communication between objects in a system happens indirectly, through a mediator object acting as intermediary. Interaction between objects is thus separated from the objects themselves and can be varied independently of those objects. The Mediator pattern is particularly useful in a system that needs to be flexible but requires many separate, yet tightly coupled objects. Localizing integration in mediator objects allows connections between objects to be easily rearranged; only the mediators need to be adjusted, not the objects themselves. Sullivan and Notkin (1996) document the use of Mediator in PRISM, a system for planning radiation treatment programs for cancer patients in which a range of separate yet interconnected components need to be kept consistent. Use of Mediator enabled the designers of PRISM to create a system that modeled real-world tight coupling between components, and yet was relatively easy to modify,

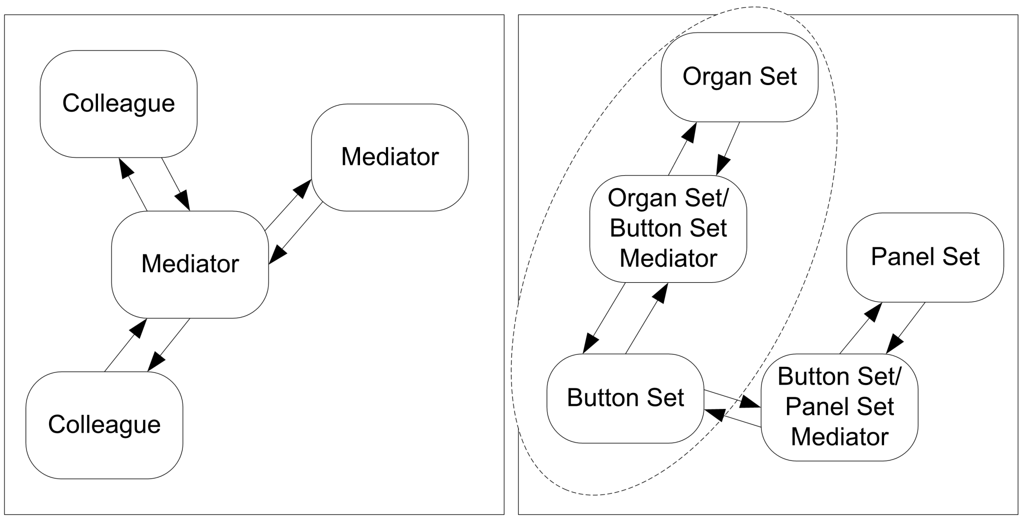

**Figure 2-3: On the left, a picture depicting the Mediator pattern (based on (Gamma *et al.*, 1995, p. 276)); on the right, circled is a mediator and two colleagues in the PRISM system.**

maintain and extend. Mediator structures integrated systems as "collections of visible, independent components integrated in networks of explicitly represented behavioral relationships" (Sullivan *et al.*, 1996). In the case of PRISM, for example, a set of tumors, a set of corresponding buttons, and a set of panel displays are all independent object sets kept consistent by mediators. The manifestation of the pattern – the mediator objects – is clearly visible; in fact, it is critical to the overall structure of the system, as shown in Figure 2-3.

The structure of the Mediator pattern is embodied in the existence of mediator objects and the kind of relationship they have with other objects. The forces Mediator resolves have to do with balancing the need for flexibility or adaptability with the need to maintain a high level of connections between system components.

Abstract Factory (Gamma *et al.*, 1993, p.87-95) allows families of related or dependent objects to be created without indicating which concrete class they belong to. A client can interact with an abstract factory that delegates object creation to whichever concrete factory is in use at that time. The particular concrete factory in use can thus be changed without affecting client code and the implementation of concrete factories can be hidden from the client. Furthermore, Abstract Factory makes it easy to ensure objects belong to a family of related products. For example, the ET++ application framework (Weinand *et al.*, 1988) achieves portability across a range of window systems by using Abstract Factory to create interactive components for the particular system in use at the time.

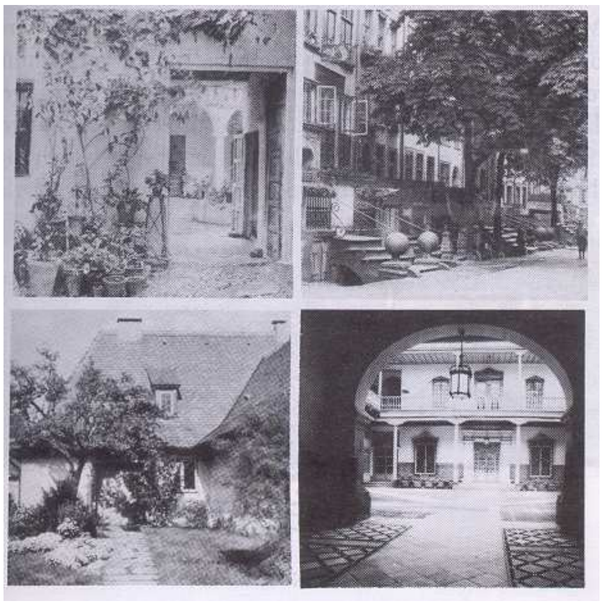

**Figure 2-4: Four photos of entrance transitions (Alexander *et al.*, 1977, p. 551).**

The structure of Abstract Factory has to do with the way it uses the inheritance mechanism to capture the commonality between related but different families of objects, and thus separates the creation of objects of particular, concrete classes from the clients using those classes. The forces Abstract Factory resolves have to do with enforcing the constraint that a family of related objects ought to be used together, and providing flexibility when it comes to deciding which particular family should be used.

### 2.2.2 Architectural Patterns

One of Alexander's patterns that he himself rates as very successful (by denoting the pattern with two asterisks (1977, p. xiv)) is Entrance Transition (1977, pp. 548-552). Entrance Transition smoothes the approach to an entrance so that the entrance is not abrupt. The transition can be built with trees, or a winding path, or arches, or just about anything, as shown in Figure 2-4.

The physical structure produced by Entrance Transition is marked by a change in light, sound, direction, or something that helps to smooth the transition from outside to inside, or street to yard, or similar, as shown in Figure 2-5. The forces resolved by Entrance Transition have to do with human psychology. Between inside and outside, for example, people tend to adjust their behavior in subtle ways. Alexander notes that at least one research study has indicated that an entrance transition provides space and time for that to happen, and thus makes people more comfortable with the change in environment and helps them adapt their behavior to be more appropriate to the new environment.

Another pattern that Alexander describes is Accessible Green (1977, pp. 304-309), illustrated in Figures 2-6 and 2-7. This pattern captures the need for many small public parks to be scattered across a city, in contrast to the common case where a few large parks are created with the intention of providing access to greenery for many people. The fact that city people want walking-distance access to small public parks is well documented. Alexander uses an informal study to measure a park's accessibility and argues that a park must be within three minutes walking distance for people to make use of it. Accessible Green thus argues that small public parks ought to be scattered at roughly 1500-foot (460m) intervals throughout a city.

The structure of Accessible Green has to do with the creation of public parks, the distance between such parks and the impact that the creation of many, small public parks has on the overall structure of a city as a whole. The forces that Accessible Green resolves have to do with the fundamental and conflicting human needs to experience nature and open spaces and to live in community, as well as forces that drive the creation of big cities.

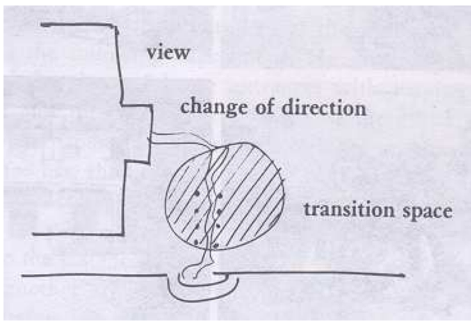

**Figure 2-5: A sketch highlighting key aspects of an entrance transition (Alexander *et al.*, 1977, p. 552).**

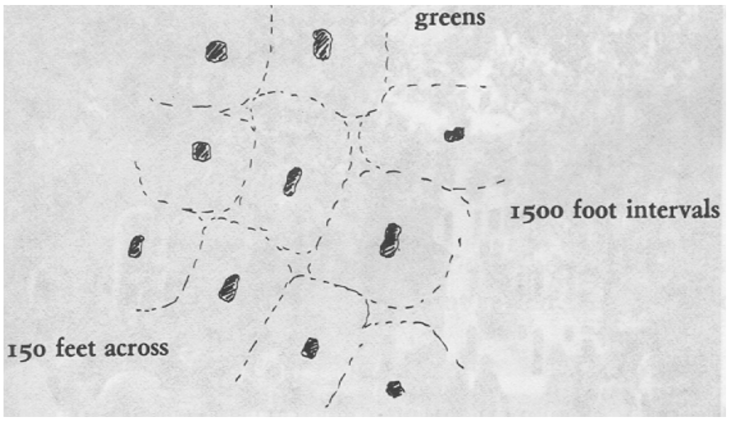

**Figure 2-6: A drawing showing the kind of structure created by accessible greens (shown as black dots), each roughly 150 feet across, scattered at roughly 1500 foot intervals (Alexander *et al.*, 1977).**

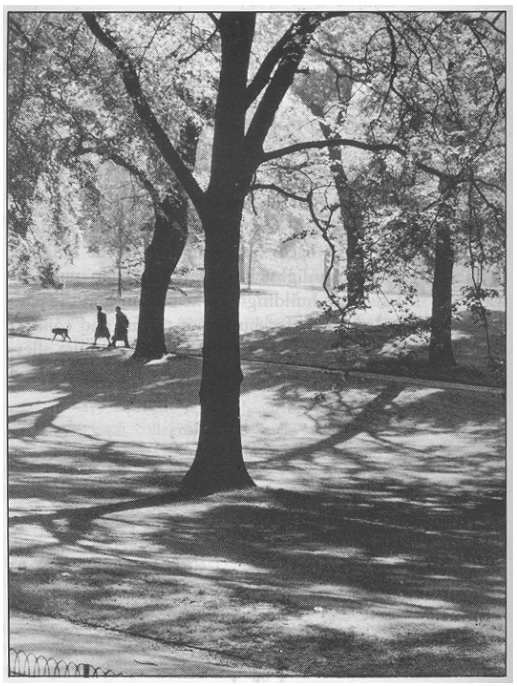

**Figure 2-7: A photograph of an accessible green (Alexander *et al.*, 1977).**

In contrast to Accessible Green, with its focus on the outdoors, Four-Story Limit (Alexander *et al.*, 1977, pp. 114-119) focuses on how high to build buildings. It argues that there is abundant evidence that high buildings have a detrimental effect on the physical and social well being of residents. Children who grow up in such buildings and live above the fourth floor are likely to have poor motor skills, for example. Adults who live in such buildings and live above the fourth floor are significantly more likely to become depressed than the average citizen, even if they have no prior history of depression. The fourth floor seems to be a critical limit because it represents about the maximum height from which, for example, parents can watch children play in the street, and people can interact with the street from a balcony or similar. For these reasons, Alexander argues that even in a big city, buildings should be built four stories high or less.

Four-Story Limit is structural in that it is not just a general rule stating that buildings should be a certain height, but defines a building structure of four stories or less. The forces that it resolves have to do with the basic human need for interaction, but the desire of or necessity for many people to live in small, self-contained residences in a large city.

### *2.2.3 Patterns in Physics and Biology*

Stewart and Golubitsky (1992) describe many patterns in natural and other physical systems: snowflake shapes, crystals and the x-ray diffraction patterns they produce, hexagonal lattices typically formed by fish territories, dew drops forming on a spider web, the splash that forms when a drop hits the surface of a glass of milk, and symmetric shapes formed by chaotic processes. Unlike the domains of software and architecture, the natural, physical domain is not a domain in which artifacts are explicitly designed; patterns arise out of the processes of nature, rather than from any explicit process of design. Stewart and Golubitsky argue that the notion of pattern in nature is closely tied to symmetry; patterns are formed as a result of transformations that break symmetry. Intuitively, symmetry has to do with a correspondence in size, shape or relative position around some center or axis. When forces in a system make the existing symmetry no longer sustainable, the system adopts a new configuration in which symmetry is redistributed; this phenomenon is called symmetry breaking.

One example of a pattern highlighted by Stewart and Golubitsky is the pattern formed by droplets of water hanging on a spider web, as shown in Figure 2-8. When a droplet of water forms on a thread of a web, it doesn't spread evenly like a cylinder around the length of the thread, with the same symmetry as the thread, because the surface tension of the water compresses it along the direction of the thread. Like a very tall stack of books tied with elastic, the fully symmetric state is thus unstable,

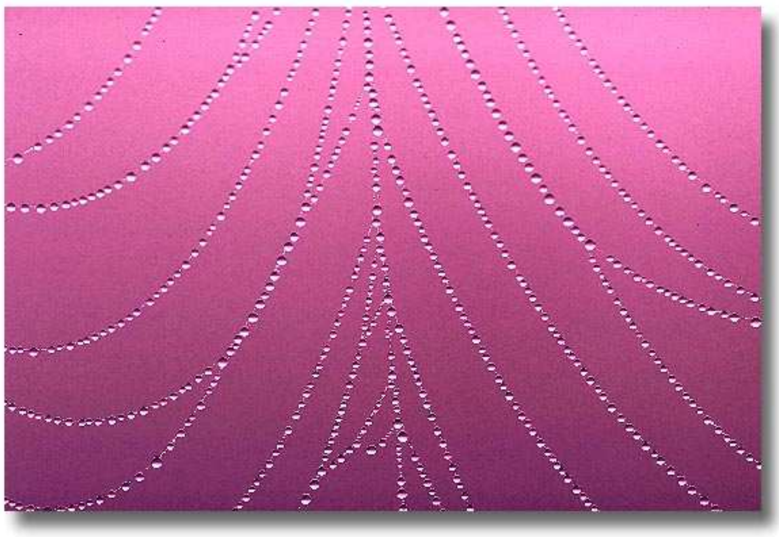

**Figure 2-8: Dewdrops forming at regularly spaced intervals on a spider web (Tonhouse, 1997-2002).**

and the water buckles at various points, forming discrete droplets. Yet, if the translational symmetry of the thread is to remain in some form, the water must buckle at regular intervals, which it does, and if the reflectional symmetry of the thread is to remain in some form, then the dewdrops must be bilaterally symmetric, which they are. The original thread had full translational symmetry; it could be divided by straight lines into a sequence of identical threads of any given length. The thread with dewdrops has translational symmetry at regular intervals; it can be divided by straight lines into a sequence of identical threads of a length appropriate to the spacing of the droplets. Similarly, the original thread had full reflectional symmetry; it could be divided at any point and one side of the thread would be the mirror image of the other (assuming infinite length). The thread with dewdrops has reflectional symmetry at regular intervals; it needs to be divided at midpoints between droplets for both sides of the thread to look the same. The thread with dew drops has some, but not all, of the symmetries of the original thread, and no new symmetries; the symmetries of the thread with dew drops are a thus a *subgroup* of the symmetries of the original thread.

Another pattern Stewart and Golubitsky highlight is that of a droplet falling into a glass of milk and making a splash, as shown in Figure 2-9. Before falling into the glass, the droplet is round and thus has rotational symmetry: it is symmetric for any rotation about its center. When the droplet falls into the milk, forces, such as the surface tension of the milk, mean that it cannot maintain its round shape. The new shape that forms is a crown-like shape that no longer has full rotational symmetry. The actual crown that forms has a subgroup of full rotational symmetry; it has some but not all of the same symmetries as the drop, and no new symmetries.

All of the patterns that Stewart and Golubitsky describe have structure, and that structure is tied to their geometry or symmetry. For example, the structure of the thread with dewdrops can be defined in terms of the translational and reflectional symmetries of the thread and its segments. Those symmetries have a particular, structural relationship to the symmetries of the original thread. Similarly, the structure of the crown-shaped splash is defined by its rotational symmetries and constrained by the shape of the original droplet.

### *2.2.4 Patterns in Other Domains*

One domain in which a wide variety of researchers have noted the importance of patterns is that of organizational systems. Harrison and Coplien (1996) build on the work of many researchers to assemble a collection of key patterns defining the organizational structures of software development enterprises. They analyze the social networks of the enterprises, identifying patterns of organization and process that characterize high productivity.

For example, Work Flows Inward notes that those directly involved in producing software, such as designers and coders, should be consumers rather than producers of communication in the organization. Communication and hence work should flow inward; it should flow inward to software producers, rather than outward from them. Architect Also Implements is another pattern characteristic of productive software enterprises. A project architect is typically responsible for providing overall, guiding technical direction for a project. If the architect guides but does no coding, it causes two problems: the architect's thinking is limited to abstractions which often gloss over important subtleties, and the architect can be perceived as being out of touch with the realities of programming. A project architect should therefore participate in implementation.

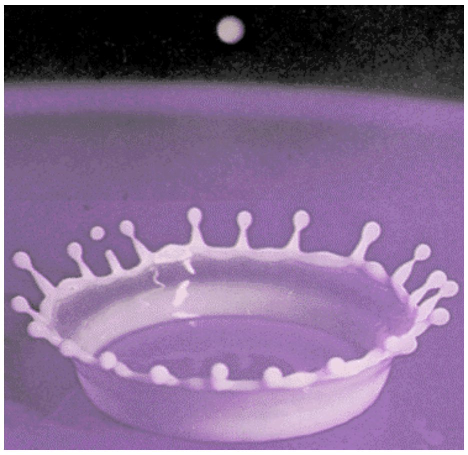

**Figure 2-9: A milk drop and the splash it makes falling into a glass of milk (Edgerton, 1957).**

Each of these patterns is structural. Work Flows Inward creates structure defining the way that communication takes place, and Architect Also Implements defines aspects of the structure of the relationship between architect and coders. The absence of either of these patterns changes the structural relationships of an organization. Work Flows Inward balances forces such as the need for developers to have time to code and the need for client input to be communicated to developers. Architect Also Implements balances forces to do with the need for strong interconnections between participants in a software development project, the need for overall technical direction, and the need for leaders to be in touch (and perceived to be in touch) with the concerns of a wide range of people.

Another domain in which many patterns have been explicitly documented is that of pedagogy. Based on his experience of teaching computer science, Bergin (2000) documents a number of patterns, many of which apply to pedagogy in general rather than to computer science in particular. For example, Spiral suggests introducing two mutually dependent concepts by alternating explanations at gradually increasing levels of detail. Each concept is presented first at an overview level, then at a slightly more detailed level, and so on. Early Bird suggests that the most important ideas in a course should be presented early, and returned to repeatedly, so that they can be reinforced and their importance made clear.

The structure of Bergin's patterns has to do with the course structure that they combine to produce. Each pattern defines a certain element of overall course structure. The forces resolved by Spiral relate to how students learn; in a context where many topics are interrelated, students need to have a basic understanding of many topics in order to be able to solve interesting problems. A detailed explanation of one topic that does not allow them to solve interesting problems leaves students bored. Early Bird resolves forces to do with students tending to remember best what they learn first and students needing to be able to relate more detailed explanations to a bigger picture.

Based on discussions with former Chicago White Sox second baseman Jack Perconte, Coplien (1999) documents a collection of T-ball patterns that contribute toward building the dynamic structure of a T-ball swing. Step Toward Pitcher points out that to transfer as much energy as possible to the ball when hitting, a batter must put their full weight into a hit. They can help achieve this by stepping directly toward the pitcher when hitting. Shift Weight Forward gives direction as to how a swing should end. The batter's front foot should be firm and next to the plate, and only the toe of the back foot should be touching the ground, supporting no weight.

Each of the T-ball patterns is structural, building a part of the dynamic structure of the T-ball swing. Step Toward Pitcher resolves forces to do with the batter needing to step forward to get full power in a swing, but needing to step forward in the right direction in order to maximize that power. The forces that Shift Weight Forward resolves have to do with the batter needing to start a swing on the back foot, but needing to finish it on the front foot, in order to generate the necessary power in the swing.

Eric Bern (1964) studied the psychology of human relationships and documented games that articulate recurring patterns of relationship between people. One such game is occasionally played between therapist and client and is called Indigence. Indigence is typically played by social workers who agree to help a client provided the client does not get better, and by clients who agree, for example, to look for work, provided they are not required to find any. A related game is I'm only Trying to Help You, played by social workers who offer suggestions to clients that are supposedly helpful, but in fact, as both worker and client are aware, are likely to fail. The client and worker both use the suggestions to blame the other for failure and avoid really trying to find a solution to a difficult problem.

Bern's games are pattern-like entities that resolve forces to build a structure. The structure they build is that of a particular kind of psychological relationship and the forces they resolve have to do with allowing people not to face fundamental fears. For this reason, the patterns Bern documents are usually ultimately destructive.

## 2.3 What is a Pattern Language?

While the term "pattern language" is used in a variety of ways (Angluin, 1980; Noble, 1998a; Alexander *et al.*, 1977), by far the most predominant use of the term is to refer to a collection of patterns that work together to build a system. Software developers are only just beginning to realize the power of pattern languages for solving complex problems and building systems. An isolated pattern solves an isolated design problem; a pattern language shifts the designer's thinking to the systems level (Coplien, 1996, pp. 17-21):

> A pattern language is a collection of patterns that build on each other to generate a system. A pattern in isolation solves an isolated design problem; a pattern language builds a system.
>
> (Coplien, 1996, p. 17)

> A collection of patterns forms a vocabulary for understanding and communicating ideas. Such a collection may be skillfully woven together into a cohesive "whole" that reveals the inherent structures and relationships of its constituent parts toward fulfilling a shared objective. This is what Alexander calls a *pattern language*.
>
> (Appleton, 2000)

The term "pattern language" is most widely used in the software and architecture domains. Amongst those in the software patterns community, exactly how patterns work together and precisely what kind of collection of patterns constitutes a pattern language has been a matter of debate for some time. Some key points of consensus have, however, begun to emerge, and these are discussed in Section 2.3.1. Alexander (1977; 1979) put considerable effort into defining his understanding of the concept. His notion of a pattern language has become by far the most widely used in architecture, and has formed the basis for ideas of pattern languages developed in many domains, including software. Alexander's work is outlined in Section 2.3.2. Outside the software and architecture domains, some uses of pattern language reflect an Alexandrian influence, while others do not. Section 2.3.3 discusses how the term is used in different domains.

### *2.3.1 What is a Pattern Language: a Software Perspective*

Fincher (1999) points out that scale is the fundamental organizing principle of pattern languages. The way that a language captures "the shape of the whole into which the pieces fit" is by indicating relationships between patterns of different levels of scale.

Larger-scale patterns rest on smaller-scale patterns, and smaller-scale patterns contribute to larger-scale patterns. In other words, a pattern language must contain patterns of different levels of scale.

Buschmann *et al.* (1996, pp. 11-15) also recognize the importance of scale to pattern languages[1]. They identify three different levels of scale for patterns in software pattern languages: architectural, design, and idiomatic. Architectural patterns relate to fundamental, overarching structural schema according to which software systems are organized; examples include Pipes and Filters and Blackboard. Design patterns relate to the subsystems of a software architecture, describing recurring structural relationships that solve a general design problem; examples include Proxy and Mediator. Idioms are low-level, language-specific patterns that show how to implement particular aspects of subsystems using a particular programming language; the Counter Pointer idiom for C++ programmers is one such example. Buschmann *et al.* argue that a software pattern language ought to tie patterns of each of the above different levels of scale together. The language ought to describe how its constituent patterns are interconnected and work together.

Coplien (1998a; 1998b; 2000a) emphasizes that pattern languages are structural. A pattern language is not simply a guide to good design. It expresses key, structural relationships that characterize systems built with the language. The fact that scale is the key organizing principle of pattern languages, though, does not mean that the structure of a pattern language is a simple, tree-like hierarchy. As Coplien (1996, p. 17) notes, "[a] pattern language is not just a decision tree of patterns", because containment, or specialization is not the only kind of structural relationship between patterns. The structure of a pattern language is more complex than a simple, tree-like hierarchy and can be better represented by a directed, acyclic graph.

Some collections of patterns, such as the Gang of Four collection (Gamma *et al.*, 1995) have been called pattern *catalogs*. Salingaros (2000) notes that a pattern language is more than just a pattern catalog. A catalog can be likened to a dictionary. Just as an English dictionary provides little information about the grammatical rules governing the combination of words to form sentences, a pattern catalog provides little sense of how to effectively combine patterns, and why certain combinations are

[1] The authors actually use the term pattern system, instead of pattern language, but the meaning is essentially the same. They prefer to avoid the use of the word language, arguing that a pattern language must be computationally complete (that is, the patterns in a language must cover every aspect of importance in a particular domain), whereas a pattern system allows for gaps or blanks in the collection.

effective or valid, and others not. A pattern language, on the other hand, tells the designer which patterns to combine, and how to combine them, to form a higher-level pattern.

Salingaros further points out that a pattern language has the properties of an emergent system; the combination of smaller-scale patterns produces new and unexpected properties not present in the constituent patterns. These properties are expressed in a higher-level pattern. Coplien (1996, p.17) recognizes that the emergent properties of a pattern language make it much more powerful than a pattern in isolation. Both a pattern and a pattern language generate structure, but a pattern language has the power to generate a complex system, rather than simply to solve an isolated design problem.

## 2.3.2 What is a Pattern Language: an Architectural Perspective

Most researchers who discuss software pattern languages draw on architect Christopher Alexander's (1977; 1979) foundational work on the theory of pattern languages for inspiration and insight. Alexander describes a pattern language as a structural network of inter-related patterns of different levels of scale:

> The structure of a pattern language is created by the fact that individual patterns are not isolated.
>
> …
>
> Each pattern then, depends both on the smaller patterns it contains, and on the larger patterns within which it is contained.
>
> …
>
> Each pattern sits at the center of a network of connections which connect it to certain other patterns that help to complete it.
>
> …
>
> And it is the network of these connections between patterns which creates the language.
>
> (Alexander, 1979, pp. 311-313)

The interconnections between patterns are determined by the patterns themselves:

> The elements [of a pattern language] are patterns. There is a structure on the patterns, which describes how each pattern is itself a pattern of other smaller patterns. And there are also rules, embedded in the patterns, which describe the way that they can be created, and the way that they *must* be arranged with respect to other patterns.
>
> (Alexander, 1979, p.185)

And the power of a pattern language comes from the ever-changing nature of the patterns and their interconnections:

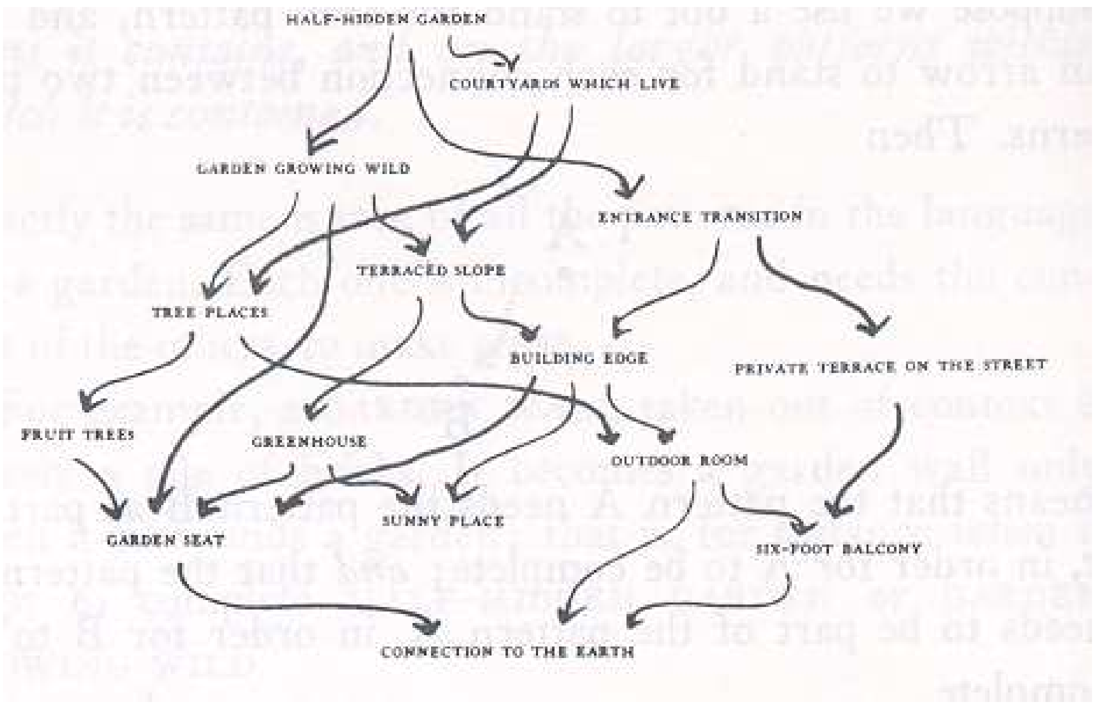

**Figure 2-10: Alexander's (1979, p. 314) diagram of a language generating a half-hidden garden.**

> … we tend to think of patterns as "things," and keep forgetting that they are complex, and potent fields.
>
> Each pattern is a field – not fixed, but a bundle of relationships, capable of being different each time that it occurs …
>
> A collection of these deep patterns, each one a fluid field, [is] capable of being combined, and overlapping in entirely unpredictable ways, and capable of generating an entirely unpredictable system of new and unforeseen relationships.
>
> When we remember this, it may be easier to recognize how powerful they [patterns] are – and that we do indeed, have our creative power as a result of the system [language] of patterns that we have.
>
> (Alexander, 1979, p. 223)

The structure of a pattern language is often represented as a diagram consisting of text representing pattern names and arrows between pairs of patterns representing structural connections in the language between those patterns, as shown in Figure 2-10. Alexander describes the kind of connections between the patterns as follows:

> Suppose we use a dot to stand for each pattern, and use an arrow to stand for each connection between two patterns. Then [A• →• B] means that the pattern A needs the pattern B as part of it, in order for A to be complete; and that the pattern B needs to be part of the pattern A, in order for B to be complete.
>
> (Alexander 1979, p. 313)

Alexander argues that the fundamental relationship between patterns is one of *differentiation*: smaller patterns differentiate larger ones. He informally defines differentiation as smaller patterns building on larger patterns by a process of elaboration, rather than one of combining pre-formed parts. To Alexander, the relationship between patterns of different levels of scale in a language is like the process by which an embryo develops: a whole gives birth to its parts by splitting. Of course, some system transformations also involve an actual, concrete aggregation of matter: Alexander describes this as *piecemeal growth*. Piecemeal growth describes the process of adding new structure to a system in a way that is incremental and seeks to mold new structure so that it fits well with the overall existing structure. Differentiation and piecemeal growth are thus the core principles underlying Alexander's concept of how the patterns in a pattern language build on each other.

Directly related to Alexander's fundamental concepts of differentiation and piecemeal growth is generativity. As Coplien (1996, p. 32) points out, Alexander argues that complex problems can often not be effectively solved by direct attack. Good solutions to complex problems are achieved by an indirect approach that provides the right kind of environment for a problem to resolve itself over time as a system slowly evolves:

> … This quality in buildings and in towns cannot be made, but only generated indirectly by the ordinary actions of the people, just as a flower cannot be made, but only generated from the seed.
>
> (Alexander, 1979, p. xi)

Alexander compares the generativity of a pattern language to that of a spoken language:

> An ordinary language like English is a system which allows us to create an infinite variety of one-dimensional combinations of words, called sentences.
>
> …
>
> A pattern language is a system which allows its users to create an infinite variety of those three dimensional combinations of patterns which we call buildings, gardens, towns.
>
> …
>
> … Thus, 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, pp. 185-186)

Alexander uses the term pattern language in two related but different senses. The first sense is to refer to a particular collection of patterns embodied in, or used to generate, a particular building. The second sense refers to a more general language from which a variety of buildings (or, a variety of pattern languages in the first sense of the term) can be built. Alexander argues that more general languages can be generated from a collection of specific languages, for particular buildings, by identifying common patterns across those languages, and by recognizing underlying similarities between patterns in different languages that can be expressed as higher-level patterns.

In some ways the two, seemingly different senses in which Alexander uses the term pattern language are in fact the same if each language is viewed at the right level of scale. A more general language is then simply a specific language but on a much larger level of scale. Once a more general language is written down for general use, as Alexander himself has done (1977), any recorded structural ordering of the patterns in the language can only be a rough guide, because that ordering is in part determined by context. Alexander, for example, divides his two hundred and fifty-three patterns into only three levels of scale. In a pair of patterns that are similar in scale, either one may be above or below the other in the language; context will determine what is appropriate.

When it comes to judging the quality of a pattern language, Alexander argues that two characteristics are of primary importance; a good language is both morphologically and functionally complete. A language is morphologically complete when the patterns in the language together form a complete structure, with no gaps. It is functionally complete when the patterns in the language allow the forces they deal with to resolve themselves (Alexander, 1979, pp. 316-317).

### 2.3.3 What is a Pattern Language: Other Perspectives

Salingaros (2000) is a physicist who has studied Alexander's work as part of his own work on pattern languages and their structure. Salingaros argues that patterns exist in all areas of human life and cannot be dictated or forced, but arise out of use and are accepted on their benefits. Pattern languages are significant because they capture the behavior of complex systems and arise out of two different needs: the need to understand and control a complex system, and the need for design tools that facilitate functional and structural coherence. Salingaros further argues that a pattern language is itself a complex system and therefore its structure is that of a complex system. The structure of a pattern language is therefore hierarchical; different processes occur at different levels of scale, and connections exist both within and across levels. The structure of a whole pattern language cannot be defined simply from its constituent patterns; rather, a pattern language is an emergent system:

> Each level in a complex hierarchical system is supported by the properties of the next-lower level. The combination of patterns acting on a smaller level of scale acquires new and unexpected properties not present in the constituent patterns, and these are expressed in a higher-level pattern. Patterns on higher levels are therefore necessary because they incorporate new information.
>
> (Salingaros, 2000)

Salingaros notes that complex systems are essential to human life because they "[facilitate] human life and interactions, and [have] to continually stand up to tests of their efficacy in this respect". The patterns of complex systems are not dictated or forced, but "[arise] out of use, and [are] accepted on [their] benefits". Pattern languages are thus incompatible with stylistic rules, which tend to be based on fashion and rhetoric, and have no connection to human needs. Stylistic rules are like viruses; their success is measured not by how well they serve humanity, but by how many copies of them are produced. They are just images with a superficial symbolic content, such as "flat, smooth, continuous walls at street level", and tend to suppress natural complexity. Individuals who propagate such stylistic rules can be seen as agents for replicating viruses, in the sense that they destroy the complex connectivity that promotes life.

Another characteristic of pattern languages highlighted by Salingaros is that while a language needs a certain degree of internal consistency, it's external connectivity is much more important than its internal consistency. This is both because the complexity of the system it represents may act against internal consistency, and because it is the interaction between many, distinct individual systems that allows complex system behavior to emerge. As Alexander (1988) also notes, the structure of a language is more complex than can be represented in a simple tree-like hierarchy.

With respect to the evolution and repair of pattern languages, and the needs that drive pattern language development, Salingaros makes two important points. In terms of evolution and repair, he argues that a new, innovative pattern is superior only if it increases connectivity of the pattern it is replacing with the majority of surrounding patterns. In terms of the needs that drive the development of pattern languages, he argues that there are two fundamental, but very different needs. One of those is the need to understand and possibly control a complex system, and the other is the need to create complex systems that are functionally and structurally coherent.

The most widespread use of the term pattern language that does not make reference to Alexander relates to Angluin's (1980) work on "Inductive Inference of Formal Languages from Positive Data". Angluin's domain is that of natural languages, and she defines a pattern as "a concatenation of constants and variables" and a pattern language as "the set of strings obtained by substituting constant strings for the variables". Angluin characterizes the conditions under which it is possible to infer a family of pattern languages from positive data. The sense in which she uses pattern language is notably different to Alexander's usage of the term. In particular, Angluin defines a pattern language as a collection of concrete expressions of a more general rule, whereas a language in Alexander's sense expresses the more general rules of how the components of a system work together. Perhaps the most relevant aspect of Angluin's work for this project is the comment she makes about how to determine whether underlying structure can be correctly inferred from positive data:

> Gold (1967) gives a definition of correct inductive inference in which, informally speaking, the inferring process is presented successively with a larger and larger corpus of examples, eventually including any particular example, and simultaneously makes a sequence of guesses of the underlying rule being exemplified. If the sequence of guesses eventually converges to a single value which is a correct description of the underlying rule the inference is correct. If the sequence of guesses oscillates indefinitely, or converges to an incorrect description, the inference is incorrect.
>
> (Angluin, 1980)

## 2.4 Examples of Pattern Languages

An important principle of software pattern communities worldwide is that patterns and pattern languages are best understood in the context of concrete examples, and the following sections each focus on examples from a particular domain. Section 2.4.1 details software examples. The particular software examples chosen – the C++ idioms language (Coplien, 2000a) and the CHECKS Pattern Languages of Information Integrity (Cunningham, 1995) – are chosen because they are well-known and well-regarded languages each addressing a very different software problem. For these reasons, they are also used extensively in Chapter 5 as examples. The examples in Section 2.4.2 are drawn from Alexander's (1977) architectural language for towns, buildings and construction. Examples in Section 2.4.3 focus on languages in domains other than software or architecture and serve the purpose of demonstrating the multidisciplinary nature of the pattern language concept.

### 2.4.1 Software Pattern Languages

The C++ Idioms Pattern Language (Coplien, 2000a), based on idioms drawn from a textbook of programming idioms (Coplien, 1992), guides the building of an inheritance hierarchy in C++, with particular focus on algebraic types. Key issues in building such a hierarchy include efficient and effective memory management, placement of algebraic operations in the hierarchy, and implementation of abelian operations without using multiple inheritance. The base patterns in the language are Handle/Body and Concrete Data Type. Handle/Body splits a class into interface (handle) and implementation (body) classes, providing the convenience of scope-based memory allocation where the entity concerned is not in (program) scope. Concrete Data Type shows how to effectively manage memory allocation and deallocation when object lifetime does not follow program scope. Other patterns build on these base patterns to address further issues of memory management. For example, Counted Body adds reference counting to a Handle/Body context, and Handle/Body Hierarchy focuses on efficient and effective use of Handle/Body within an inheritance hierarchy. Detached Counted Body adds reference counting to a Handle/Body context in the case where the body code cannot be modified. Envelope/Letter modifies Handle/Body by making the Body class inherit from the Handle class, in addition to the Handle class maintaining a pointer to the Body class. Changes to the Handle class are then propagated to the Body class by definition, rather than needing to be explicitly forwarded. Further down the language, Algebraic Hierarchy outlines how to distribute algebraic operations in a Handle/Body-based hierarchy of algebraic types. The patterns below Algebraic Hierarchy enable heterogeneous addition to be implemented within such a hierarchy. The approach used is for the *add* method in each subclass to deal only with operands of its own subclass (Homogeneous Addition), for each subclass to know how to promote itself to the next class up the hierarchy (Promotion Ladder), and then for operands of different type to be promoted to the same type before adding (Promote and Add, NonHierarchical Addition). Type Promotion indicates how to handle built-in types.

Each of the patterns in the idioms language generates some kind of structure in the form of a relationship between objects in a C++ program. The structure of the language itself is shown in the language structure diagram in Figure 2-11. The diagram highlights the fact that the structure of the language is more complex than a simple, tree-like hierarchy, as some patterns build on more than one other pattern. Promotion Ladder, for example, builds on both Envelope/Letter and Algebraic

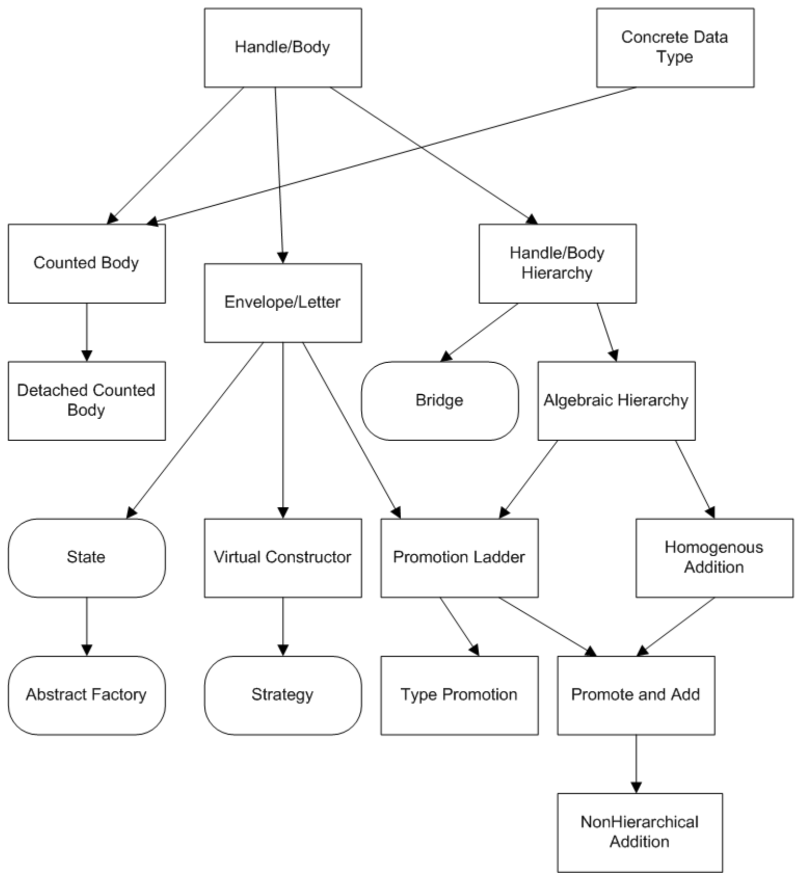

**Figure 2-11: A diagram showing the structure of the idioms language (based on (Coplien, 2000a, p. 169)).**

Hierarchy. A programmer needing to modify an existing C++ program can use the language to help determine how best to modify that program to achieve the desired result, and what the consequences of any modifications will be. For example, by splitting an object into two objects linked through a pointer, Handle/Body can make inheritance less useful; Handle/Body Hierarchy addresses this issue.

The CHECKS Pattern Language (Cunningham, 1995) guides the development of software that accepts user input. The language shows how to separate good input from bad and how to minimize the amount of bad input recorded. It consists of ten patterns relating to quantifying the domain model, providing feedback about the domain model to the user as transparently as possible, and addressing the long-term integrity of information. Whole Value, Exceptional Value and Meaningless Behavior relate to quantifying the domain model. Whole Value argues that specialized values that are meaningful in the domain should be constructed and used for message arguments and as input/output units. For example, the duration of a contract should be specified in units of weeks or days, rather than as an integer. Not every user input, however, will fit into the range of attributes covered by Whole Value. For example, if "agree" and "strongly agree" are two typical responses to a query, an answer such as "illegible" cannot be quantified in the domain model.

Exceptional Value addresses this problem. It suggests that "one or more distinguished values [be used] to represent exceptional circumstances", rather than extending the range of attributes covered by Whole Value. Exceptional Value incorporates into the domain model a place for missing data that may appear at a later stage. Where no condition can be anticipated under which data might have domain meaning, Meaningless Behavior can be applied. Meaningless Behavior suggests that data without potential domain meaning should be ignored: methods should be written "without concern for possible failure", and "the input/output widgets that initiate computation [should be expected] to recover from failure and continue processing. The user will interpret blank output as meaning that inputs are not applicable and/or outputs are unavailable."

Five patterns, Echo Back, Visible Implication, Deferred Validation, Instant Projection, and Hypothetical Publication provide feedback to the user about the domain model. The need to Echo Back information is driven by the use of Whole Value to construct appropriate values for the domain model. Echo Back requires that any information that the user enters is displayed back to them, so that the user is aware of transformations made to the data entered. For example, the date entered as "6/8/00" might be echoed back as "06/08/2000", or the date "3/1/00", entered as a pay date, might be echoed back as "3/3/2000" if the system records pay on Fridays. Visible Implication combines Echo Back with the displaying of computed information. It points out that values derived or computed from data entered by the user should be displayed back to the user along with the values corresponding to the data entered. For example, if the user enters a quantity of "12" and a unit price of "$4", then a total on hand value of "$48" might be computed. Displaying the computed value can increase the user's awareness of how the system works and make the visual review more effective. Deferred Validation suggests that the validation provided by Whole Value and Echo Back is sufficient step-by-step validation, and detailed validation of user input can be delayed until particular checkpoints are reached. Instant Projection extends Visible Implication and represents a shift in focus from data entry to publication. It provides the ability to project the consequences of a publication of data before that publication is made. Hypothetical Publication adds an extra layer of data checking on top of Deferred Validation. It allows for publications to be released internally prior to actual publication, providing extra opportunity for error detection, which could be particularly important where publication is high-risk.

Forecast Confirmation and Diagnostic Query address the long-term integrity of information. Forecast Confirmation identifies the need for “a mechanism for adjusting and confirming values associated with mechanically published events”. Since computer models often lag behind real-world events, “up-to-date” computer models are often mechanically generated. When the computer system eventually catches up with real-world events, though, the mechanical model must be adjusted to ensure accuracy. For example, a mechanically posted deposit might slightly differ in amount from the actual deposit. Diagnostic Query allows every value in the system to be traced so that the user can see, step by step, what transformations have occurred on raw input data to produce given output. In a context where Whole Value and Exceptional Value are being used, Diagnostic Query makes visible to the user useful information that may have been accumulated by using these two patterns, but is not visible in the form output is displayed on the user interface.

Each pattern in the CHECKS language is connected to other patterns in the language, and those connections are clearly more complex than simple containment. For example, Exceptional Behavior and Meaningless Behavior are each connected to Whole Value; each addresses an aspect of quantifying the domain model based not addressed by Whole Value, but based on the assumption that Whole Value is being used. Whole Value and Exceptional Value represent alternative solutions to the problem of bad data; it is context that determines which is more appropriate to use. Echo Back addresses a problem partly created by the use of Whole Value; input data may be transformed to an appropriate domain model value, and the user needs to be aware of any such changes. The CHECKS language generates the structures of particular user interface programs, and the language is itself structural, in the sense that there are structural relationships between the patterns in the language. The connections between patterns provide the programmer with an understanding of the structures of the system they build, and therefore can help the programmer to see how to add to or modify the program when necessary.

### *2.4.2 Alexander's Pattern Language for Architecture*

Alexander (1977) developed a pattern language for the architecture of towns, buildings, and construction. His largest patterns are for regions and towns. As patterns become gradually smaller in scope they relate to neighborhoods, building clusters, buildings, and alcoves. Those patterns smallest in scope present details of construction. For example, Subculture Boundary is a larger-scale pattern that creates a physical, shared space between areas of different subculture in a highly populated area. Accessible Green is slightly smaller in scale than Subculture Boundary. It notes that access to green, open space is critical for people's well being, but that people will not walk more than about three minutes to access such space. Small areas of green, open space should thus be evenly distributed across a city. Positive Outdoor Space is smaller in scale than Accessible Green, and notes the importance of giving defined shape to outdoor space, rather than it being just the leftover pieces around buildings and other structures. Tree Places is again smaller in scale, and notes that trees ought to be planted with regard to the kind of spaces they can form - enclosures, avenues, squares, and groves, for example. Garden Wall is of a similar level of scale to Tree Places. It highlights the importance of some kind of wall around a garden to provide relief from noise in a busy area. Seat Spots is smaller in scale than both Tree Places and Garden Wall. It guides the positioning of outdoor seats to maximize their usefulness.

The patterns in Alexander's language are not isolated entities. Each pattern can only be fully defined in the context of other patterns. For example, the larger-scale Subculture Boundary is incomplete unless it contains Accessible Green. An Accessible Green is itself incomplete without Positive Outdoor Space, Tree Places, and a Garden Wall. Seat Spots elaborates one way of creating Tree Places. A Garden Wall is also needed by other larger-scale patterns, such as Quiet Backs. Alexander's patterns thus build on each other to generate architectural forms through the processes of differentiation and piecemeal growth. Differentiation is about smaller-scale patterns combining to form larger ones; smaller-scale patterns add detail to the structure of larger-scale ones. Piecemeal growth recognizes that the language itself is in a constant state of flux, as redundant patterns are dropped and new patterns are added to the language.

The structure of Alexander's pattern language is more complex than that of a simple hierarchy. Many smaller-scale patterns complete more than one larger-scale pattern. For example, Quiet Backs points to the need for buildings in busy, noisy areas to have alleys behind them with a calmer, quieter atmosphere. Garden Wall not only helps to complete Accessible Green, but also helps to complete Quiet Backs.

Sometimes, the connections between patterns are contextual; a pattern is needed by another pattern, but only in a certain context. For example, Private Terrace on the Street needs some kind of wall to be complete, but depending on context, any of Sitting Wall, Half-Open Wall or Garden Wall could be most appropriate. Alexander's language suggests the kinds of situations in which particular kinds of wall are most appropriate; that is the guidance that it gives the designer. In the end, though, the particular choice of which kind of wall to use in a particular situation rests with the designer. Alexander's language thus highlights important structural constraints, but emphasizes the human nature of design.

### 2.4.3 Pattern Languages in Other Domains

One of the most comprehensive collections of pattern languages is the collection of four languages that Harrison and Coplien (2004) have developed relating to different aspects of organizational structure. They point out that an organization has structure; it is a social system that may be more or less hierarchical, authoritarian, and governed by top-down control. Differences in organizational structure lead to different processes. The Linux culture, for example, is a very shallow and broad hierarchy, in contrast with classic military organizations whose hierarchy is narrower and much more overt. The pattern languages developed by Harrison and Coplien each have structure, and in each the constituent patterns combine to form a whole entity: a system. The patterns combine in a number of ways. For example, in the Project Management Pattern Language, the pattern Programming Episode suggests breaking down coding for a large project into manageable chunks by coding those decisions that can be made at any point in time, and not focusing on those that can not. Someone Always Makes Progress outlines the importance of one person always making progress on the primary task. The connections between patterns in the languages suggest a number of patterns that can refine or elaborate on Someone Always Makes Progress, and can help tailor it to a particular situation: Developing in Pairs, Team Per Task, Sacrifice One Person, and Day Care. Developing in Pairs enables one person to be always at the keyboard, Team Per Task assigns one team to a major diversion, so that other teams can more easily focused on the primary task, Sacrifice One Person assigns "only one person to the distraction", and Day Care separates training from programming tasks, leaving experienced programmers free to code rather than training less experienced programmers.

In the domain of pedagogy, Anthony (1996) presents a pattern language for teaching difficult technical topics in a classroom setting. Her patterns include Mix New and Old, which highlights the importance of intermingling review and new work, Chicken and Egg, which suggests a way to effectively explain two concepts that are each a prerequisite of the other, Quiz Games, which lessens the boredom of doing review work, and Colorful Analogy, which suggests explaining important, detailed, and possibly boring concepts by using an interesting analogy. The patterns work together to generate both individual classes and a complete course. They provide insight into both the overall structure of the course and the processes used to create that structure. Mix New and Old, for example, working together with Quiz Games and Colorful Analogy, shows how to produce an interesting and effective learning session made up of both review and new work.

Coplien (1999) has published a pattern language that generates a T-ball swing. Some patterns focus on preparation to swing; examples include Cover the Plate, which explains how to position oneself at the correct distance from the plate, and Head Towards Home Plate, which suggests the batter tip their head forward toward home plate to improve balance. Other patterns are more dynamic, such as Step Toward Pitcher, which points out that the step the batter takes in swinging should be toward the pitcher, and Shift Weight Forward, which explains how the batter can shift weight toward the ball as they swing, thus increasing the quality of the hit. Combining the patterns in the language in the right way generates the dynamic structure of a T-ball swing. How to combine the patterns is not explicitly articulated, but some clues can be found in the content of the patterns themselves.

Hiroshi (2001) has published a pattern language for reading books to children. The language includes patterns such as The Favorite, which notes that while adults dislike repetition, children love it, and reading the same story night after night is not a bad thing. Wings of Fantasy reminds adults that picture books are, for children, a gate to a different world. Read Slowly points out that reading slowly is part of enabling children to enter the different world that a picture book opens to them. The patterns in the language combine to create the structure of a space and time that is special for parent and child.

## 2.5 Defining Key Elements of a Pattern

As Alexander (1979, p. 247) notes, a pattern provides a contextualized solution to a problem. A successful pattern writer must understand the appropriate *context* of a pattern, what *problem* the pattern addresses, together with the *forces* that define the problem and must be resolved to bring about a *solution*.

Context is inherently part of a pattern because the recurring solution captured by a pattern tends to recur in specific, non-arbitrary situations. A pattern's context defines the kinds of situations in which the pattern will occur or is likely to be relevant, and specifies factors that, if changed, would invalidate the pattern (Coplien, 1996, p. 8).

As discussed in Section 2.3, a pattern language provides context for a pattern by locating the pattern in a particular position in the overall structure of the language. Each pattern helps to complete larger-scale patterns, works together with patterns of a similar level of scale, and is completed by smaller-scale patterns (Alexander *et al.*, 1977, p. xii).

The problem addressed by a pattern represents a state or source of difficulty that needs to be resolved. Patterns do not address trivial sources of difficulty, but, rather, key, challenging problems in a particular area – problems that designers in that area face time and time again, in one form or another. By working together in a language, patterns "capture" a domain: "together, the patterns in a language identify the domain's key concepts and the important aspects of their interplay" (Winn and Calder, 2002).

In physics, a force can be defined as "the influence that produces a change in a physical quantity" (Lessico Publishing Group, 2004). The use of the term force with respect to patterns arises out of the architectural heritage of patterns:

> A building architect designs arches and walls to balance the forces of gravity with the forces from adjoining structures, so the structure is balanced and centered. Balanced forces support a firm, structurally sound system. In software, we use the term *force* figuratively because there are rarely physical forces we must balance. Even Alexander used the [term] force in a figurative sense, particularly as he was concerned about balancing the forces of human esthetics and comfort with the physical structure of a town or building.
>
> …
>
> (Coplien, 1996)

Coplien (1996, p. 9) points out that "[forces] determine why a problem is difficult". It is easy to define what is wrong with a bad design, but much harder to characterize good design. Different forces may pull a solution in different directions and those forces will need to be balanced in a good solution. Within a pattern language, each pattern represents a cluster of forces that is highly interactive; the interactions between forces within a pattern should be much higher than those across patterns.

The solution provided by a pattern proposes an answer for the particular problem being addressed by resolving existing, possibly conflicting forces. The solution is structural; it shows how to generate a stable structure that holds the appropriate forces in tension. Within a pattern language, the solution of a given pattern generates new, unresolved forces that become the basis for new problems to be addressed by smaller-scale patterns. The solution also provides the basis for the context of smaller-scale patterns.

# Chapter 3

# Software Design

This chapter places the work of this thesis in the context of other approaches to software design. Section 3.1 compares patterns and pattern languages to other well-known techniques and tools used in software design. Section 3.2 describes existing work within the software patterns community that relates to how patterns work together and how to precisely define the kind of relationships that exist between patterns.

## 3.1 Patterns and Software Design

As well as patterns, software developers have many other design tools available to them. Section 3.1.1 compares patterns to other related tools, highlighting differences and similarities between them. But patterns and, most especially, pattern languages are not simply tools that can be applied in random ways in software development. A pattern is "both a process and a thing" (Alexander, 1979, p.247) and as such embodies some notion of software development process (Lea, 1994). For this reason, it is worth comparing pattern-based software development to approaches based on other techniques. Section 3.1.2 and subsequent sections each focus on one software development approach and highlight how it is different from or similar to a patterns-based approach.

### *3.1.1 Patterns and Other Design Tools*

Viljamaa (1995) provides a useful starting point for distinguishing patterns from a range of other design tools commonly used in software development:

> Patterns are related to but different from: paradigms, idioms, principles, heuristics, architectures, frameworks, role-models.
>
> You could say that a paradigm is a very abstract pattern, or style of work that can be followed consistently throughout the system. Idiom is a language-specific typical way of using and combining elementary building blocks. Principle is an invariant that can hold globally, or always; it could be a synonym for design rule. Heuristics aid decision making, without claiming absolute goodness for the actions suggested. Heuristics could be used to choose among multiple alternative patterns. Architecture refers to the total structure of an application, possibly described by the multiple patterns involved. Patterns have been called micro-architectures. Frameworks refer to collections of concrete classes working together to accomplish a given parameterizable task. Role-models describe a single coordinated collaboration among multiple participants (the framework classes can serve in multiple roles simultaneously). Role-models may be the closest thing to the formalization of patterns.

Many existing design tools are subtly different from patterns. In more detail, differences between patterns and those tools highlighted by Viljamaa are as follows:

- A *paradigm* is a conceptual framework that shapes the fundamental structures of a domain, whereas a pattern is a transformation on those structures.
- *Idioms* are reusable expressions from a given programming language. They are not part of the standard language syntax, but are fundamental building blocks for programmers building programs. For example, Coplien (1992) points out that C++ programmers should provide classes that include certain characteristics, even though the syntax of the language does not require it. The connection between idioms and patterns is that both represent recurring structures that solve recurring problems, but idioms are generally limited to one language, whereas patterns are more general. Idioms are thus like technology-specific patterns.
- *Principles* and *heuristics* are guidelines for creating solutions, whereas patterns are actual, structural solutions to problems.
- A *software architecture* is defined by Shaw and Garlan (1996, p. 1) as "the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns." A given software architecture can be characterized by idiomatic patterns, but the term software architecture refers to larger-scale system structure and behavior, whereas patterns occur and work together across all levels of scale.

- A *framework* is a collection of concrete classes that work together. Wirfs-Brock (1990) describes a framework as “a skeleton implementation of an application or an application subsystem in a particular problem domain.” A framework thus provides code that can form the basis for a number of different systems. When code is added to a framework, though, the framework is not thought of as being modified by that context. Instead, the framework is only the skeleton, and is added to differently each time it is used. Further, whereas a framework is a codified architecture, implemented in a particular language, patterns are “ways of *using* a programming language” (Gamma *et al.*, 1993). A framework can be thought of as similar to a pattern language, because it describes the structure of a whole system, but the key difference is that the framework is only a skeleton, whereas a pattern language represents a flesh-and-bones system.
- *Role modeling* is a way of describing object collaboration. In contrast to a class diagram, which specifies the classes to which particular objects belong, a role-model shows how objects behave, according to the role types of the roles they are playing (Riehle and Gross, 1998). Role models focus on the behavior of objects in a system, whereas patterns focus on system structure and how that structure evolves over time.

A commonly used design tool not mentioned by Viljamaa is a *template*. A template is like a mold, or an example that is worth emulating. It is more rigid than a pattern; even though it can have parameters of variation, those parameters are clearly defined and do not shape the template in any significant way. In contrast, context shapes patterns; it does not alter key structural relationships, but it does change structure.

Some authors have confused templates with patterns. For example, Coad (1995) focuses on object-oriented design patterns and argues that patterns are in fact templates that express commonly used object relationships. The kind of commonly used relationship Coad refers to can be illustrated by analyzing point-of-sale systems. Such systems often require objects that represent various kinds of participants, and various kinds of transactions. A customer and a cashier are each typical participants, and an order and deposit are each typical transactions. Participants tend to have certain kinds of attributes, such as number and password, as do transactions, which have a date, time and status. Participant objects tend to be related to transaction objects in a one-to-many relationship: one participant may make any number of transactions. Coad thus defines a *Participant-Transaction* template that consists of an outline for defining participant and transaction objects in a one-to-many relationship, including typical object attributes and services, and listing example participants, transactions, and other templates with which this template might connect.

While relationship is central to patterns, it is not sufficient to define them. A pattern is a solution to a problem *in a context* (Alexander, 1979, p. 247). Coad's templates lack contextual information and are thus not patterns, although they are potentially useful for those using object-oriented design methods. Context is vital in that it shapes and defines a pattern. Berczuk (1995), in his review of Coad's book, makes a similar criticism when he points out that Coad's "patterns and strategies do not provide guidance for their use". They lack information about when to apply the pattern, and why, and are thus little more than algorithms, lacking the "power of patterns to encapsulate and share design expertise".

Another commonly used tool not mentioned by Viljamaa is *architectural styles*. Architectural styles represent large-scale patterns of software systems, similar to those found in Buschmann's (1996) study of patterns in software architecture. Architectural styles describe recurring structures in software architectures. A system's software architecture characterizes its overall structure, instead of focusing on the details of which algorithms and data structures are used for computation. Shaw and Garlan (1996, p.1) list issues that comprise the global system structure such as component composition, global control structures, communication, synchronization, data access protocols, distribution of functionality among design elements, and more. They further point out (1996, p. xi) that software architectures are characterized by what they call architectural styles, which they describe as "patterns for system organization that software developers use purposefully but nearly unconsciously." By documenting these patterns, Shaw and Garlan seek to provide a common vocabulary for a large body of implicit knowledge, making that knowledge more easily accessible. They define an architectural style as consisting of a collection of components and connectors, and a set of constraints on how they can be combined. For example, Pipe and Filter is one architectural style that uses filters as components and pipes as connectors. Each filter has a set of both inputs and outputs, and specifies particular criteria specified according to which input data is examined for possible output. Each pipe sends the outputs of one filter to the inputs of another. Filters are independent and do not share state with or know the identity of other filters. The correctness of a pipe and filter system should not depend on the particular order of pipes and filters.

The key difference between architectural styles and patterns is that architectural styles tend to refer only to large-scale system structure, whereas patterns occur across all levels of system scale, and patterns of different levels of scale work together to generate a system.

While patterns can be compared to other design formalisms, techniques, and tools, they are more than just one of these. For example, patterns embody both a particular kind of solution structure and a process; they embody transformations on structure. The use of patterns has implications for overall design process, structures, and even the values on which development is based. Alexander argues that the patterns in a pattern language must be applied in order of morphological importance (1979, p. 381-382), and that the kind of transformation a pattern makes on a system is to differentiate existing structure (1979, pp. 365-384). The process of development, then, begins by considering overall structure, and smaller-scale structure must be evaluated for its effect on the whole. Patterns are applied in sequence (Alexander, 1979, p. 373), with each pattern addressing one key problem (Winn and Calder, 2002). Design is therefore an incremental process, based on small-step change.

Lea (1994) affirms that the use of patterns in software development implies that changes to a system are made in small steps. He also notes that patterns affirm the human-centered nature of development as a social process. For this reason, a patterns-based approach to software design requires that users participate in the design process and roles in a design team are integrated so that, for example, an architect is also sometimes a builder. These values raise challenges for modeling software development:

> Pattern-based design activities resist accommodation within a linear development process, and raise challenges in the construction of suitable process models that still meet costing, predictability, and control criteria.
>
> (Lea, 1994)

Coplien (1996, pp. 34-45) emphasizes similar values to Lea. He points out that patterns capture proven, real solutions, rather than postulates, so use of patterns implies a design process grounded in existing, real solutions rather than a particular theory. Coplien also points out that ethics and interdisciplinary scope are central to the use of patterns.

### 3.1.2 Task-Oriented Abstract Algorithms

Harandi and Young (1998) have developed a standard set of skeleton algorithms that can be used as a starting point for software development. They point out that software developers "… use skeleton algorithms when developing a mental model of a program". In other words, developers initially think in terms of a rough outline that may omit important information affecting final code. Further, in order to effectively maintain and reuse software, a software developer needs access to design decisions and information about why they were made as well as actual code, because code itself simply does not contain enough information. A useful task from a software development point of view is to develop, from the great variety of informal skeleton algorithms used by programmers, a standard set of skeleton algorithms for reuse in software development. Harandi and Young describe such a set of task-oriented abstract algorithms (TOAAs) together with a system for making use of them, including automating documentation of aspects of the design process.

There are many similarities between TOAAs and patterns. TOAAs play a role similar to that of larger-scale patterns in that they allow for a broad, outline view of a problem to be articulated and for details to be filled as they are worked out. TOAAs thus encourage small-step development. Developing a standard set of TOAAs fulfills a similar purpose to the development of a pattern language that is built by observing common patterns across a range of development projects, and TOAAs recognize the importance of context in design, by including discussion about why design decisions were made.

Perhaps the most important difference between TOAAs and patterns relates to pattern languages. Patterns in a pattern language are of many different levels of scale, and both Alexander (1979) and Salingaros (2000) emphasize the importance of having enough levels of scale, and of patterns of different levels of scale working together. TOAAs do address some issues of scale. For example, a TOAA includes a list of implementation decisions that can act as a record of the transformation of the algorithm from abstract algorithm to concrete implementation. In Harandi and Young's work, however, the emphasis on the importance of levels of scale for solving complex problems is significantly less than that found in pattern languages.

### 3.1.3 Refactoring

Refactoring is a technique for improving the internal structure of code without changing its external behavior. It uses small, known, incremental changes to reduce the short-term cost of making the effort to write good code. Fowler (1999, p. 53) defines a refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”.

Sample refactorings include *Extract Method*, *Move Method* and *Replace Temp with Query*. *Extract Method* can be used, for example, in the case where an existing method is too long and potentially confusing. It suggests finding a code fragment whose lines seem to hang together and form a group, and turning that fragment into a method with an appropriate name. *Move Method* can be used to transfer a poorly placed method from one class to another. The old method is replaced by a simple delegation. *Replace Temp with Query* suggests using a method call instead of a temporary variable to obtain the result of an expression. Temporary variables can only be seen in the context in which they are used, and thus tend to encourage longer methods. Each refactoring in Fowler’s (1999) book is explained using several examples of varying degrees of difficulty, and the motivation for the refactoring explained. The mechanics of the refactoring are also discussed, highlighting potential pitfalls if the extraction is done badly, and suggesting how to do the refactoring well so as to avoid those pitfalls.

Refactoring is similar to pattern-based approaches in several ways: it is a design technique grounded in actual, designed artifacts; individual refactorings are explained in the context of examples; contextual information about why the refactoring is useful and potential pitfalls when applying it are highlighted; refactoring focuses on small-step, piecemeal growth; and refactoring is based on a view of the design process whereby design occurs continuously during development, rather than a good design being a necessary precursor to good code.

Refactoring also differs from patterns in several ways. Perhaps the most important difference is that refactoring focuses on applying particular refactorings, rather than how different refactorings fit together to provide a broader picture of software design. Whereas patterns in a language build on each other to generate a system, collections of refactorings tend to lack a notion of how they work together to generate a larger whole. Refactoring also explicitly distinguishes between adding functionality to code and rearranging existing code, and defines refactoring as the latter and not the former.

### ***3.1.4 Unified Modeling Language (UML)***

The Unified Modeling Language (UML) (OMG, 1997-2003) is a notation that "helps you specify, visualize, and document models of software systems, including their structure and design", prior to investing considerably more time and money in actually building the systems. UML facilitates functional, object, and dynamic models of systems:

- A functional model is one that describes what the system can do, usually from the user's point of view. In UML, use cases are used to represent functionality. An object model describes system structure in terms of "objects, attributes, associations and operations" (Bruegge and Dutoit, 2000, p. 24).
- An object model describes relationships between objects. In UML, an object model is described with class and object diagrams. A class diagram shows the static relationship between classes, whereas an object diagram shows how objects created at runtime are related to those classes.
- A dynamic model describes a system's internal behavior. In UML, sequence diagrams describe behavior in terms of a sequence of messages passed around amongst a collection of objects, statechart diagrams describe behavior in terms of individual object states and transitions between those states, and activity diagrams describe behavior in terms of control and data flow, showing how the execution of a set of operations triggers the execution of another set of operations.

UML is a notation deliberately designed to be separate from any particular software development methodology, and so it does not define a standard development process (Larman, 1998, p. 19). Reasons for this include a perception that UML would be more widely accepted if it remained independent of a particular development methodology, and that appropriate methodology varies considerably according to context. Larman (1998, p. 20) points out, however, that UML does make assumptions about general principles underlying good design process. In particular, design consists of multiple cycles of a plan, build, and deploy development sequence. Recently, developers of UML have developed the Rational Unified Process (RUP) (Kruchten, 2000). RUP uses UML extensively in its implementation and Krutchen (2000, pp. 28-29) describes it as a guide for effective use of UML. RUP is an iterative and explicitly evolving software development process centered around five views of a project's architecture: the use-case, logical, implementation, process, and deployment views (Reed, 2002, p. 22-23). The use-case view describes system functionality as perceived by external "actors", and also system requirements. The logical view focuses on how system functionality is delivered. The implementation view describes implementation modules and dependencies between them. The process view shows how the system can be divided into processes and processors, and the deployment view shows how the system is readied for and put into action.

Many authors who discuss the use of UML in software design also discuss using UML and patterns together. Larman (1998, p. 4) argues that UML facilitates documentation of software design by providing a way to document important design information such as how objects interact and which classes have which responsibilities. He argues that because patterns capture best practice in a design domain, they can be used to help make good choices about interactions and responsibilities, and can be documented using UML. Larman describes a pattern as a general principle or idiom guiding software development, and argues that once such a principle is described in a problem and solution format, and given a name, it can be called a pattern (1998, p. 189). Savitch (2004, p. 617) describes patterns as "design outlines that apply across a variety of software applications". He argues that UML is a useful formalism for describing design patterns. Cooper (2000) uses UML diagrams to illustrate patterns for Java programmers. Most discussions of patterns in the context of UML limit the concept to GoF-like (Gamma *et al.*, 1995) design patterns and furthermore, they rarely discuss the pattern concept in any depth. The way that patterns combine to generate solutions to complex problems is rarely discussed; instead the focus tends to be on individual patterns as isolated solutions to particular problems.

There are several differences between a UML-based approach and patterns. In contrast to UML, patterns make inseparable model and process; they are well recognized as being a means of describing both system structure and transformations on that structure. Even if UML does, as Larman suggests, provide guidance for some kind of underlying development process, the process it describes is different from that encompassed by patterns. In particular, using patterns does not imply having separate planning and building phases; instead, the two intertwine.

A second key way in which UML and patterns differ is in the way in which they model a system. UML models system structure by means of class diagrams. In an object-oriented software system, patterns identify recurring relationships between objects as the key elements of system structure. This difference is important because it raises a question about which system structure is important to model. In a context where change is frequent, patterns can be argued to facilitate good design by encapsulating particular concepts that are likely to be varied, thus allowing them to be varied independently of other system features (Winn and Calder, 2002).

A third difference is in the value placed on documentation. The primary purpose of documenting patterns is to share design expertise, not to document structures particular to a given system. System-specific structures are likely to change over time, and then documentation either has to be constantly updated or simply becomes inaccurate. A pattern, however, captures an invariant that is independent of any one system and thus will remain constant even when systems that use it change (Winn and Calder, 2002).

A UML-based approach to software development that incorporates RUP has interesting similarities to a patterns-based approach in that it both encourages an iterative approach to software development (Kruchten, 2000, pp. 6-8) and explicitly acknowledges the need for the software development process itself to evolve (Kruchten, 2000, p. 34). The developers of RUP deliberately deliver regular upgrades, and deliver them electronically, so as to encourage such a view. The openness of patterns similarly encourages a view of the development process itself where that process is continually evolving (Lea, 1994). The main differences between an RUP/UML and a patterns-based approach to software development lie in the kind of structure articulated by UML as compared to patterns, and in the importance placed on values (Coplien, 1996, pp. 34-45) in a patterns-based approach.

### 3.1.5 Extreme Programming (XP)

Extreme Programming (XP) is a lightweight software development methodology centered on flexibility and adaptability, and designed for small to medium-sized software projects. Proponents argue that XP is challenging the traditional waterfall model of the software development process. They argue that XP's commitment to continually embracing change makes for a more accurate model of the software development process than a rigid model that suggests that requirements are developed prior to system design, which happens prior to coding, and so on.

Beck (2000) explains that XP rests on four values: communication, simplicity, feedback, and courage. It seeks to manage four key project variables: cost, time, quality, and scope. It provides structure for the basic development activities: coding, testing, listening, and designing. And it is underpinned by fifteen key principles, which include the five fundamental principles:

- *Rapid feedback* argues that quick feedback is the most effective.
- *Assume simplicity* maintains that the simplest design that works is the most efficient.
- *Incremental change* states that too much change at once is ineffective.
- *Embracing change* says that options should always be kept open as much as is possible while still solving the current problem.
- *Quality work* argues that quality is the least flexible of the four project variables and must be high.

XP is further characterized by twelve practices. If a software project uses more of these practices, or uses them to a higher degree, it is considered better. Examples of the twelve practices are as follows:

- *The Planning Game* refers to determining the scope of the next release quickly by combining business and technical estimates, updating the plan to reflect reality.
- *Small Releases* is about quickly developing a simple system, putting it into production, and maintaining a short cycle for new releases.
- *Metaphor* suggests that rather than talking about software architecture, a shared story should form the basis for understanding how a system works as a whole.
- *Simple Design* argues that the system should continually be kept as simple as possible and extra complexity removed if noticed.
- *Pair Programming* refers to the practice of production code being written in groups of two programmers, with each group at one machine.

Lippert *et al.* (2003) have used XP with extensions to develop complex software projects. They argue that XP is “a disciplined approach to software development that emphasizes customer satisfaction and teamwork”, and that XP can, if adapted appropriately, be effective for large as well as small development projects. They point out that while developers tend to find the extra freedom offered by XP attractive, managers tend to believe it to be unplanned and uncontrollable.

Extreme programming is similar to patterns in its lack of slavish adherence to formal methods, its blending of human and technical issues, and its emphasis on small-step change. It differs from patterns in that, although lightweight, it is explicitly a software development methodology. Furthermore, it lacks the structural focus of patterns and pattern languages and does not explicitly take the context of the development task into account.

### 3.1.6 Agile Development

Agile development is a software development methodology arising out of extreme programming and other such lightweight methodologies. It draws together commonality in lightweight development methodologies into a "Manifesto for Agile Software Development" (Beck *et al.*, 2001). The manifesto makes a statement about the following comparative values:

- Individuals and interactions rather than processes and tools.
- Working software rather than comprehensive documentation.
- Customer collaboration rather than contract negotiation.
- Responding to change rather than following a plan.

The manifesto argues that even though the values on the right are important, those on the left are more important.

Williams and Cockburn (2003) note that another characteristic of agile methodology is recognizing that software development is an empirical rather than a defined process. An empirical process is one which, however well defined, is unlikely to lead to a predictable outcome because of unpredictable changes that take place while the process is being carried out. It requires "short 'inspect-and-adapt' cycles and frequent, short feedback loops". In contrast, a defined process such as assembling an automobile is one that produces the same results every time. Current issues in agile development include how to scale it beyond the small to medium-sized projects for which it is currently known to work well, how to blend agile development with more plan-driven approaches to development, and how to smoothly bring about changes in decision-making authority and accountability implied by a shift to agile development.

Agile development is similar to patterns because it is not a plan-driven approach, it is characterized by short feedback cycles, it has a human-centered approach to development, and it insists that design and implementation are inseparable activities. It differs from patterns for the same reason that XP does; its focus is not on generating complex system structures, but on more accurately modeling existing practice in software development.

### 3.1.7 International Organization for Standardization (ISO) 9000

A standard can be defined as "an acknowledged measure of comparison for quantitative or qualitative value" (Lessico Publishing Group, 2004). ISO develops standards for industry, business and government organizations. Those standards seek to establish fair competition among diverse suppliers from many countries, within a free market economy. ISO 9000 (International Organization for Standardization, 2004) is a series of international quality standards for manufacturing and service organizations. The series is primarily focused on quality management, and aims to prevent defects in the production of goods by planning and application of best practices at every stage of business. The ISO 9000 standards identify basic disciplines and general criteria by which an organization can ensure that products meet customer requirements. Within the ISO 9000 series, the ISO 9001 standard applies to engineering disciplines, and together with a special set of guidelines (ISO 9000-3) can be applied to the software process (Pressman, 1997). ISO 9001 delineates twenty areas of requirements definition, the first five of which are management responsibility, quality system, contract review, design control, and document and data control. For a software organization to register as ISO 9000 compliant, it must first develop policies and procedures addressing each of the twenty requirements and then be able to verify, by means of an independently conducted audit that those policies and procedures are being adhered to.

The ISO 9000 approach to software development has little in common with a patterns-based approach. The ISO 9000 approach is focused on reducing the risk of developing a poor quality product, whereas a patterns-based approach seeks to build a high quality product. In a patterns context, quality is about good structure, whereas in ISO 9000 terms, quality is about lacking defects and demonstrating a required degree of functionality. An ISO 9000 approach assumes that the chances of developing a quality product can be improved by specifying universally defined processes and procedures that control and verify design and design changes. A patterns-based approach is driven more by systems thinking (Senge, 1990) and compared to ISO 9000 is less prescriptive, more informal, and more focused on generating good structure by understanding interrelationships between key system elements.

## 3.2 Analyzing Relationships Between Patterns

The patterns community recognizes that solving many complex problems requires understanding both key patterns and the way those patterns interact to generate system behavior (Coplien, 1996, pp. 32-33). Many different theoretical underpinnings for patterns, pattern languages, and connections between patterns are being explored. Some of these are limited in focus to object-oriented design patterns, which represent only a small percentage of published software patterns. Such approaches are discussed in Section 3.2.1. This limited focus is, however, in contrast to Alexander's more broadly focused theory. Approaches to understanding how software patterns work together that are broader in scope include Meszaros and Doble's classification of common relationships between patterns, Buschmann *et al.*'s pattern system based on different levels of scale, and Coplien and Zhao's symmetry-based analysis of patterns. Such approaches are discussed in Section 3.2.2, except for Coplien and Zhao's work, which is discussed in Chapter 4, reflecting the importance of concepts of symmetry breaking to this thesis.

### *3.2.1 Pattern Relationships: Object-Oriented Design Patterns*

Zimmer (1995) notes that few publications have adequately addressed issues relating to how the patterns in the Gang of Four (GoF) catalog (Gamma *et al.*, 1995) work together. He classifies the relationships into three main categories. X *uses* Y occurs when one subproblem addressed by X is similar to the problem Y addresses. *Variant of* X *uses* Y is the same as X *uses* Y except that use of Y is optional; only some variants of X use Y. X *is similar to* Y refers to the case where X and Y tackle similar problems but provide different solutions.

Zimmer does not ask how or why particular patterns are connected and thus offers little more information than the GoF book itself in this regard. It is unclear why similarity is important enough to form a category of relationship, or why similarity is more important than other forms of relationship, such as some of those highlighted by Noble (1998a).

Noble (1998a) describes a classification scheme for design patterns based on three primary relationships and a number of secondary relationships. The three primary relationships are *uses*, *refines*, and *conflicts*:

> … one pattern *uses* another pattern; a more specific pattern *refines* a more general pattern, and one pattern *conflicts* with another pattern when they both propose solutions to a similar problem.
>
> (Noble, 1998a)

Noble describes the difference between *uses* and *refines* as analogous to the difference between composition and inheritance. Composition defines a new class as containing a reference to another class, connecting components together to allow object interactions, whereas inheritance refers to the definition of a new class in terms of an existing class. Noble's secondary relationships include *variant*, *combines*, *requires,* and *tiling* relationships, and *sequences of elaboration*. He distinguishes between patterns that describe variant *problems* sharing a common solution, and patterns that provide variant *solutions* for a common problem. The *combines* relationship describes two patterns working together to solve a problem not directly addressed by any single pattern. The *requires* relationship describes a situation where one pattern cannot solve a problem without another pattern already being in place. *Tiling* describes the repeated application of a single pattern, and a *sequence of elaboration* is a sequence of patterns, from smaller to larger, connected in ways described using the proposed primary relationships.

Noble's relationships may be useful for object-oriented design patterns, but he does not explore their usefulness in a broader domain. His analogy between composition and inheritance and the *uses* and *refines* relationships may be valid, but given the multidisciplinary nature of patterns, it would be surprising if the connections between them mirrored object-oriented programming relationships. Noble suggests but does not fully explore the relationship between his primary and secondary relationships, and between those relationships and Alexander's (1979) understanding of pattern interrelationships.

Noble's classification scheme is potentially a useful way to analyze, organize, and classify object-oriented design patterns. The differences between some of the relationships he identifies are, however, unclear. For example, if two patterns provide mutually exclusive or alternative solutions to the same problem (they *conflict*), this means that they are likely to both be refinements of a more general pattern, possibly as yet unspecified. Why is it then important to have *conflicts* as a primary relationship? Similarly, patterns that provide *variant* solutions to the same problem are likely to both *refine* a more general pattern.

Noble states "[the] *uses* relationship is the only explicit relationship between patterns in Alexander's original architectural patterns". Yet patterns in Alexander's (1977) language work together to generate a solution to a larger problem, which is how Noble describes his secondary *combines* relationship. In Alexander's language, smaller-scale patterns elaborate on the structure of larger-scale patterns by a process of differentiation (1979, pp. 305-324, 365-384). The process is far more complex than that of object composition and any suggestion that the patterns in Alexander's language are connected by means of a relationship as simple as Noble's *uses* relationship misses the point.

Although Noble focuses mainly on classifying interactions between individual pairs of patterns, he has also (Noble, 1998b) worked toward developing a pattern language for object-oriented design, using his classification scheme to describe how patterns in a language relate to each other. The main contribution made by Noble's pattern language is to organize object-oriented patterns on the basis of scale, with the language beginning with larger scale patterns and gradually moving to smaller scale ones. The richness of the language is, however, limited to relationships that can be expressed by the *uses*, *refines*, and *conflicts* relationships.

In other work, Noble and Biddle (2002) argue that patterns are literature and therefore amenable to analysis using semiotics, a well-established technique for studying literature and culture. Semiotics is the study of signs, where a sign is something that stands for something else. The authors argue that both patterns and their descriptions are *signs*, made up of the *signifier*, a phenomenon that expresses the sign, and the *signified*, the mental image produced by the signifier. They argue that a pattern's solution is the signifier and the pattern's problem, context, known uses, and rationale make up the signified, and then analyze key relationships between patterns in terms of semiotics. For example, a *uses* relationship occurs when one pattern needs to use another in its implementation. In this case, the signified part of each pattern will be different but their solutions (signifiers) will be related. The *alternative* relationship occurs when two patterns provide different implementations solving the same problem; the signified parts of the pattern are similar but the signifiers are different. Finally, *specialization* occurs when two patterns have similar problem, context and solution but one is more complex than the other. In this case, both the signifier and signified are similar but one pattern is more complex than the other.

While Noble and Biddle's semiotic analysis is interesting, it is limited because it focuses only on object-oriented design patterns such as the GoF patterns. The work also suggests that Alexander's (1977) pattern language has a tree-like structure. In fact, Alexander's pattern language articulates the structure of a complex system and such structure cannot be articulated as a tree (Alexander, 1988; Salingaros, 2000). Noble and Biddle present the use of semiotics as an alternative to a pattern language for organizing a collection of patterns, but do not analyze whether semiotics is capable of articulating the complex interrelationships between patterns described by Alexander (1979).

Kodituwakku (2002) argues that mathematical formalisms underlie all existing collections of patterns. He builds on Noble's (1998a) analysis of the relationship between object-oriented patterns, arguing that the relationships between patterns in a language are always *uses* relationships, and the relationships between patterns in a catalog are one of the set of *uses*, *may use*, *refines* and *variant* relationships defined by Noble. Kodituwakku argues that once the nature of the relationships between patterns is understood, the structure of a pattern language can be represented as a connected, directed graph. Once the connections between nodes in the graph are defined in mathematical terms as morphisms, which Kodituwakku does for Noble's relationships, the structure of a pattern language can be described in terms that satisfy the axioms necessary for the language to be described mathematically as a category. Similarly, the structure of a pattern catalog can be represented as a collection of connected, directed graphs and thus can be described as a collection of categories. While Kodituwakku's work does attempt to develop a mathematical basis for the structure of pattern languages, his work is based on Noble's description of pattern relationships and suffers many of the same shortcomings, as described in previous paragraphs.

Eden *et al.* (1999) define a set of common abstractions that they argue form the basis for generating the GoF patterns. They then articulate these common abstractions using the LePUS language, which is "a fragment of higher order monadic logic", with the aim of accurately and precisely specifying the relationships between design patterns and parts of design patterns. Their work enables common relationships between, in particular, objects and classes that participate in patterns to be articulated. It does not, however, provide a means for analyzing the structural transformations that take place when patterns are applied. In fact, it cannot do this because it excludes from the formalism the context in which a pattern is applied. The context is assumed to be outside the formal understanding of a pattern, rather than essential to it.

### 3.2.2 Pattern Relationships: a Broader Focus

Meszaros and Doble (1998) seek to help pattern authors write better patterns by describing, in the form of a language, recognized best practice for writing pattern languages. While much of their focus is on syntax and esthetic issues, such as what sections to include in the pattern description and how best to lay out the pattern description on a page, they do note common kinds of relationship between patterns. For example, a pattern may *lead to*, be *set up by*, *specialize*, *generalize*, or act as an *alternative* to another pattern. A pattern leads to another pattern by creating a problem that the second pattern solves. For a pattern to be set up by other patterns, those patterns must create the problem that this pattern solves. A pattern specializes a more general pattern by making it more easily applicable in a particular problem domain, or generalizes one or more patterns if it draws out common themes across domains. A pattern acts as an alternative to another pattern if it provides a different solution to the same problem. While Meszaros and Doble's categorization is brief, it is distinctive because it does not focus on object-oriented design patterns, but encompasses a broader understanding of the pattern concept.

Few authors explicitly recognize the importance of scale in the organization of patterns. One exception is Buschmann *et al.* (1995; 1996) who discuss what they term a pattern system (instead of a pattern language) for software architecture that encompasses patterns on three distinct levels of scale: architecture, design, and idiom levels. They note that their patterns are interconnected and, in particular, that connections exist across levels of scale. For example, the Composite design pattern can be used to encapsulate messages traveling through layers in a layered architecture, described by the Layers architectural pattern (1996, p. 51). Buschmann *et al.* describe a pattern system as similar to a pattern language, but less complete. They differentiate a pattern system from a pattern catalog, arguing that a pattern system articulates relationships between patterns, such as one pattern refining another, and state that a pattern system "shows how [patterns] are interwoven with each other" (1996, p.23).

The design of Buschmann's pattern system with three levels of scale presumably mirrors Alexander's (1977) pattern language for architecture, which has three distinct levels of scale: towns, buildings, and construction. Despite their implicit recognition of the importance of scale, however, Buschmann *et al.* do not provide the accompanying in-depth discussion that Alexander (1979) does of the way that patterns work together across different levels of scale. Buschmann *et al.* limit their examples of cross-scale links to analyses within individual pattern descriptions. Their work would benefit both from an overarching discussion of scale and how it relates to their pattern system, and from a detailed analysis of the architecture of an existing software system, and how patterns of different levels of scale work together in that system. Further, although they claim that a pattern system is more than a catalog, the lack of explicit discussion of issues of scale makes the pattern system they present hard to distinguish from a pattern catalog; a system is described as a collection of patterns, and no mention is made in the definition of the importance of scale. In fact, the entire introductory discussion of pattern systems neglects to highlight the importance of scale to the authors' concept of a pattern system, and does not fully analyze the way in which patterns at different levels of scale work together.

# Chapter 4

# Symmetry Breaking and Patterns

In physical systems, patterns are transformations on system structure. Stewart and Golubitsky (1992) have explored the connection between patterns and symmetry; a pattern transforms and yet preserves structure by breaking symmetry.

Intuitively, symmetry has to do with a correspondence in size, shape or position around some center or axis. For example, a shape that can be rotated and still look the same is said to be symmetric under the rotation transformation. A symmetry means that something can be changed, yet remain the same; the orientation of the rotated shape has changed but the shape still looks the same. Figure 4-1 shows that the symmetries of a perfect starfish include rotation and reflection.

When forces in a physical system mean that the system has to change, and the existing symmetry can no longer be maintained, it is redistributed; this is called symmetry breaking. For example, when a milk droplet falls into a glass of milk, the

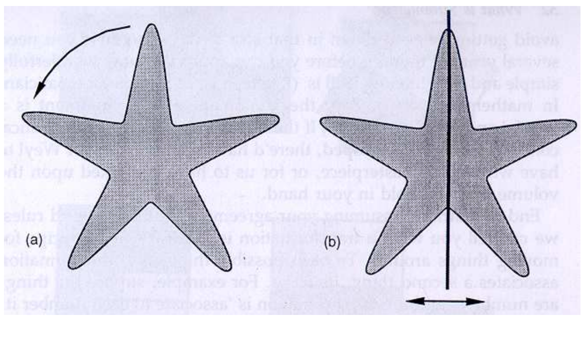

**Figure 4-1: Symmetries of a perfect starfish: (a) Rotation, (b) Reflection (Stewart and Golubitsky, 1992). The starfish looks the same under particular rotation and reflection transformations**

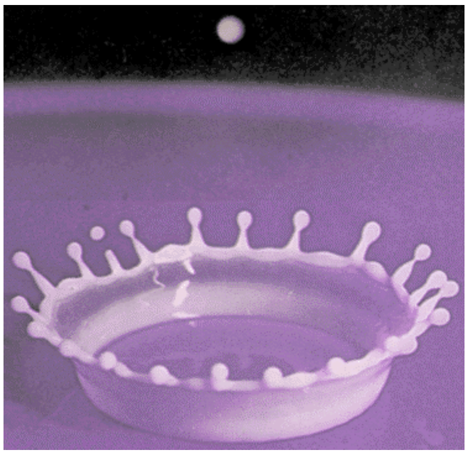

**Figure 4-2: A drop of milk splashing into a glass (Edgerton, 1957).**

splash that forms has a crown-like shape that exhibits the original symmetries of the droplet, but redistributed, as shown in Figure 4-2.

Concepts of symmetry and symmetry breaking are formally grounded in the mathematics of group theory (Ledermann, 1964; Rosen, 1995). Informal analysis of software patterns suggests that many of the best software patterns also break symmetry (Coplien, 1998a; Coplien and Zhao, 2001). A formal basis for such analysis is, however, still being developed.

This chapter outlines the concepts of symmetry and symmetry breaking on which the pattern language in Chapter 5 is based. Section 4.1 provides a basic introduction to the concepts of symmetry and symmetry breaking. Section 4.2 provides a brief overview of symmetry in a software context. A summary of both existing, informal analyses of symmetry breaking in software patterns, and initial work to establish a formal basis for symmetry in software is provided in Section 4.3. Section 4.4 uses Alexander's (2002a; 2002b) recent work on the nature of order to investigate the relationship between symmetry breaking and pattern languages.

## 4.1 Introduction to Symmetry and Symmetry breaking

Symmetry describes invariance in the face of change. A five-pointed starfish, for example, can be rotated by one-fifth of a full turn about its center and it will still look the same, as shown in Figure 4-1. The appearance of the starfish is invariant despite the rotation changing its orientation. More formally, a symmetry of an object "is a *transformation* that leaves it apparently unchanged." (Stewart and Golubitsky, 1992, p.28) What it means to be unchanged is dependent on how sameness, or equality, is defined in the given context. Further, it is not only in the context of geometric objects that symmetry can be defined. Rosen (1995, p.2) defines symmetry as immunity to possible change:

> When we do have a situation for which it is possible to make a change under which some aspect of the situation remains unchanged, i.e., is immune to the change, then the situation can be said to be symmetric under the change with respect to that aspect.
>
> (Rosen, 1995, p. 2)

Mathematically, symmetries are defined in terms of group theory. The set of symmetry transformations on an object forms its symmetry group. A group comprises a set (collection) of elements, together with a law of composition detailing how elements of the set can be combined, that satisfies the four axioms of group theory (Ledermann, 1964, pp. 2-3; Rosen, 1975, pp. 18-19):

- ***Closure***: for any two elements $a$ and $b$ of the set, their compositions $ab$ and $ba$ are also elements of the set.
- ***Associativity***: for compositions of three elements $a$, $b$, and $c$, it does not matter whether the intermediate element is associated with the first or the last, so $a(bc) = (ab)c$.
- ***Identity***: there is an element $e$ such that for every element $a$ of the set $ea = ae = a$.
- ***Inverse***: for every element $a$ of the set, there is an element called its inverse and denoted $a^{-1}$, such that $aa^{-1} = a^{-1}a = e$.

The number of elements in a group is called the order of the group. Every system thus has a symmetry group of at least order 1, since every system has at least one symmetry transformation: the identity transformation of doing nothing. Rosen thus provides the following definition of symmetry in terms of group theory:

> A system is said to have symmetry if its symmetry group is of order greater than 1. The larger the order, the higher its degree of symmetry. It is completely asymmetric if its symmetry group is of order 1, that is, consists only of the identity transformation.
>
> (Rosen, 1975, p. 22)

In the case of the starfish, its symmetry group consists of five rotations about its center (no rotation, and rotation by one-, two-, three-, and four-fifths of a turn) and five reflections about a line through the center of the starfish and each of the arms. Combining any two of these transformations simply results in one of the original ten transformations in the group. For example, rotation by one-fifth of a turn and then reflection about the vertical axis is equivalent to reflection about the upper right arm.

An important consequence of the physical laws governing how symmetry behaves is Curie's Principle, which states that when a system is transformed, symmetry is preserved; in other words, symmetry in the cause is preserved in the effect. So symmetry in nature should always be preserved. Sometimes, though, that is not what seems to happen. Consider what happens when a droplet falls into a glass of milk, as shown in Figure 4-2. The droplet of milk is the cause and the splash the effect. Before falling into the glass, the droplet has full rotational symmetry; it is symmetric for any rotation about a vertical line through its center. When the droplet falls into the milk, forces, such as the surface tension of the milk, mean that it cannot maintain its full group of symmetries. The new shape that forms – the crown-like shape – no longer has full rotational symmetry. So it seems as though symmetry has not been preserved.

In fact, Curie's principle reflects mathematical *possibility* rather than actual, physically realized states. The set of all crowns that the milk drop could potentially form when it makes a splash has full rotational symmetry. The actual crown that forms has a *subgroup* of full rotational symmetry; it has some of the same symmetries as the drop, and no new symmetries that the drop doesn't have.

Stewart and Golubitsky articulate an extended Curie principle as "a symmetric cause produces one from a symmetrically related *set* of effects" (1992, p. 58). They further point out that this phenomenon, where the actual, physically realized effect breaks the symmetry of the potential, is called *symmetry breaking*, and it leads to the *formation of patterns*:

> This paradox, that symmetry can get lost between cause and effect, is called symmetry-breaking. In recent years scientists and mathematicians have begun to realise that it plays a major role in the formation of patterns. …
>
> From the smallest scales to the largest, many of nature's patterns are a result of broken symmetry; and our aim is to open your eyes to their mathematical unity.
>
> (Stewart and Golubitsky, 1992, p.xviii)

Symmetry breaking could perhaps better be called symmetry sharing, because it refers to the way that symmetry is redistributed when forces cause it to no longer be sustainable:

> When symmetry breaks, the symmetry of the resulting state of the system is a subgroup of the symmetry group of the whole system. So symmetry-breaking is a change in the symmetry group, from a larger one to a smaller one, from the whole to a part.
>
> (Stewart and Golubitsky, 1992, p.52)

Stewart and Golubitsky argue that the formation of patterns in nature can be understood in terms of the redistribution of symmetry such that overall system symmetry will become localized when forces cause a system to change state. For example, when dewdrops form on a spider web, as discussed in Section 2.2.3 and illustrated in Figure 2-8 on page 20, the translational and reflectional symmetry of the web's thread become localized. Instead of the whole thread expressing full translational and reflectional symmetry, segments of the thread with dewdrops express this symmetry. Another example is the milk drop splash, also highlighted in Section 2.2.3 and illustrated in Figure 4-2. In this case, the full rotational symmetry of the original droplet is localized; while the crown-shaped splash is no longer fully symmetric, the points of the crown express the full rotational symmetry of the original, whole droplet.

Symmetry breaking is worth exploring as a possible underlying theoretical basis for patterns in software for several reasons: it can be described in terms of the mathematics of group theory, it offers a basis for patterns that is not domain-specific (Rosen, 1995), and it is well-recognized as underpinning patterns in physics and biology (Stewart and Golubitsky, 1992). Most attempts to describe a theory of patterns that have thus far emerged from within the software discipline have been narrow in scope, and usually limited to object-oriented design patterns, as discussed in Section 3.2. Symmetry breaking offers a perspective that has the potential to unify understanding of object-oriented design patterns and other software patterns, and place these patterns in the broader context of a multidisciplinary concept of pattern. Further, symmetry breaking is not limited to isolated patterns. Because it recognizes patterns as transformations on a larger system structure, it can also potentially be used to provide a better theoretical understanding of pattern languages and the ways in which patterns work together. Finally, systems in nature are constantly evolving and yet inherently stable: "Natural systems must be stable; that is, they must retain their form even if they're disturbed" (Stewart and Golubitsky, 1992, p.14). Developers of software systems who seek to build software systems that are stable (behave predictably) and yet able to change (to adapt in a dynamic environment) ought therefore be able to build better systems by understanding symmetry in their systems, and how that symmetry is redistributed when forces mean the system in its current form is no longer sustainable.

## 4.2 Symmetry in Software

Rosen's definition of symmetry as immunity to possible change provides a helpful starting point for understanding symmetry in software. The definition is not exclusively geometric; he simply requires that when change happens, some aspect of a situation remains unchanged. In a software context, many software structures exhibit these kinds of symmetries. A loop, for example, repeats the same code while the value of an iterator changes, and a function may be called many times, each time with different input. And while software may not be obviously geometric, it nevertheless has structure that Coplien (1998c) and Wirth (1974), amongst others, have described in geometric terms.

Zhao and Coplien (2003) define symmetry in software as "an invariant change that aims to preserve a specific property of a system". They summarize properties that symmetry might seek to preserve as follows:

1. Structure. An example of structural preservation is the notion of class. A class stipulates a uniform structure for all the objects of the class.
2. Behavior. An example of behavioral preservation is the notion of subtyping. Subtyping stipulates a uniform behavior for all the classes along an inheritance path.
3. Regularity. Symmetry considerations in operator overloading and double dispatch have the effect of preserving certain regularity. It is also true to say that structural and behavioral preservation is also about protecting regularity.
4. Similarity. Similarity is a comparison between two things. When the two things are similar, their similarity is captured as something regular. The chaos design model exhibits such regularity.
5. Familiarity. When something is familiar, it has the similarity of something else. Preserving familiarity can be called historical symmetry, symmetry of the past and the present, primitive types and user defined types.
6. Uniformity. This is just a different way of seeing regularity.

(Zhao and Coplien, 2003)

Dijkstra (2000-2001) noted the importance of symmetry to problem solving in computer science as early as the 1960s, when he used the recognition of symmetry between a keyboard and computer to solve a key problem:

> I resumed my ponderings about the synchronization between independently clocked entities, and to begin with I did so, true to my past experience, in the context of a central processor coupled to all sorts of peripheral gear, and I still remember my excitement about an in retrospect totally trivial discovery.
>
> I recall the coupling between the computer and its output typewriter with the output register between the computer's A-register and the typewriter decoder, where the computer would be temporarily frozen if it tried to execute the next type instruction before the execution of the previous type instruction had been completed. The exciting discovery was one of symmetry: just as the computer could be temporarily frozen

> because the typewriter wasn't ready for the next type action, so could the typewriter be temporarily frozen because the computer was not ready yet for the next type action; just as the computer could have to wait until the output buffer was empty, the typewriter could have to wait until the output buffer was filled.
>
> To see the precious milliseconds being lost because the expensive computer was being delayed by a cheap, slow peripheral was most regrettable, but when that same cheap, slow typewriter was idle because there was nothing to be typed, we could not care less. The excitement came from the recognition that the difference in emotional appreciation was irrelevant and had better be forgotten, and that it was much more significant that from a logical point of view they were two instances of the same phenomenon. In retrospect, this insight should perhaps be considered more radical than it seemed at the time, for decades later I still met system designers who felt that the maximization of the duty cycle of the most expensive system components was an important insight.
>
> With the symmetry between CPU and peripheral clearly understood, symmetric synchronizing primitives could not fail to be conceived and I introduced the operations P and V on semaphores.
>
> (Dijkstra, 2000-2001)

In several other talks given over the last few decades, Dijkstra (1985; 1986; 1996) has highlighted the importance of symmetry to various problems in computer science. In the 1985 talk, he noted that the whole of computer science, and especially the field of distributed computer systems, had suffered drastically because of attempts to impose a causal hierarchy on “events that constitute a computational process”. The imposition of a causal hierarchy is problematic because it completely hides “the symmetry between the sending and the receiving of messages, and between input and output”. At a luncheon speech in 1986, Dijkstra pointed out that in mathematical reasoning, recognizing helpful symmetries could reduce existing arguments/proofs by an order of magnitude. In 1996, Dijkstra analysed a variant of this well-known problem: “Two husbands and two wives have to cross a river in a boat which can only hold two people. How can they cross so that no man is in the company of a woman unless his wife is also present?” Dijkstra’s analysis noted that one effective way of solving the problem is to restore symmetry between couples, and simply regard the problem as about husbands and wives in general, rather than particular husbands and wives.

Wirth (1974) does not explicitly mention the importance of symmetry, but his advice on how to build reliable software development has a strong focus on structure. In 1974, Wirth argued that computer programming had moved on from an era where the programmer’s primary task was to formulate machine code algorithms that ran as efficiently as possible. The programmer’s task had increased in size and complexity to a point where reliability was the key challenge. Wirth argued that structure is a key tool for managing complexity. In order to be reliable, code must be “well-structured”, and in order to write well-structured code, the programmer needs to take a systematic, stepwise and structured approach to software design. Programs should be developed in small steps, and structured in blocks or modules, where larger blocks build on smaller ones, and each block has a clearly defined entry and exit point. For example, Wirth outlines the stepwise development of a sequential merge sort. He first describes the sort process as the merging of individual components into ordered pairs, then pairs into quadruples, and so on, until an entire sequence is sorted. Wirth then visualizes the sort process as the repeated transfer of tuples across two halves of an array, with the tuples doubling in size each time. The code is then written as a loop, where each time through the loop represents the completion of one step in the sort process – say, from doubles to quadruples – and the direction of transfer of sorted tuples – from either the bottom half of the array to the top, or top to bottom – is changed each time through the loop. More detailed code is then added within the overall structure of the loop.

Each step in Wirth’s design and implementation of the sequential merge sort breaks the symmetry of the previous step; that is, each step builds on the existing symmetries created by how the problem is viewed or structured at the previous step. Initially, the problem is viewed as one of sorting pairs, then quadruples, and so on. Structuring the array as two halves allows the implementation to reflect the initial view of the problem; each half of the array in turn acts as the holding bay for the next stage of the sort. Using the programming structure of a loop again reinforces this structure, by placing in the code a repeating structure that mirrors how the problem was initially stated. Further development of the code works within the loop structure, rather than compromising it. At an informal level, it is thus easily argued that each step of Wirth’s stepwise development reinforces, rather than discards, the symmetries of the previous stage. Each step represents a view of the problem that can be understood on its own, but is elaborated on by further steps. The stepwise nature of Wirth’s development, with more detailed steps elaborating on more general ones, and the relatively independent nature of each step highlight similarities between Wirth’s work and pattern languages; patterns need to be as independent as possible, and the context of a pattern defines its entry point and the resulting context its exit point.

More recent research (Gent and Smith, 2000; Fahle *et al.*, 2001; Crawford *et al.*, 1996) has focused on the importance of symmetry breaking as a tool for narrowing the search space in a combinatorial search. Various techniques exist to help recognize symmetrically equivalent searches and eliminate the need to explore the same essential search state on both sides of a choice point. For example, Gent and Smith (2000) suggest adding symmetry-breaking clauses at choice points, whereas Crawford (1996) implements such clauses prior to searching.

## 4.3 Symmetry Breaking and Software Patterns

Software patterns represent tried-and-true solutions to recurring problems. Many software patterns may also break symmetry and thus show how to transform software and maintain stability when forces, such as a need for greater efficiency or a change in requirements, mean that the software needs to change. Analyzing symmetry in software, and how best to redistribute it in accordance with the symmetry breaking seen in nature, may give clues as to how to change software and yet maintain its stability.

In the software domain, unlike in nature, there is as yet no formal basis for symmetry-based analysis; that is, there is no established analysis connecting symmetry observed and defined at an informal level with the formalized notion of symmetry, symmetric transformations, and symmetry groups in mathematics. Any attempt to analyze symmetry in software must thus be either informal or attempt to establish such a formal basis.

Coplien and Zhao have pioneered symmetry-breaking analysis of software patterns. Coplien (1998a; 1998b) has informally analyzed symmetry in several software patterns, including Bridge and Half-Object Plus Protocol (HOPP). Sections 4.3.1 and 4.3.2 provide an overview of the analyses. Coplien and Zhao's (2001) study of symmetry breaking in software patterns represents ongoing work that aims to establish a formal basis for symmetry in software; it is the subject of Section 4.3.3.

### *4.3.1 Bridge*

The Bridge pattern (Gamma *et al.*, 1993, pp.151-161) allows an abstraction to be decoupled from its implementation so that the two can vary independently. Consider, for example, developing a hierarchy involving classes for real and complex numbers. In mathematics, a complex number is a generalization of a real number, so a type representing Real should inherit from type Complex. But having Real implemented as a subtype of Complex would mean that each Real number has an unnecessary imaginary part, because the inheritance transformation comprises both interface and implementation. The Bridge pattern shows how to separate the interface and implementation inheritance so that the implementation of a class is not tied to its

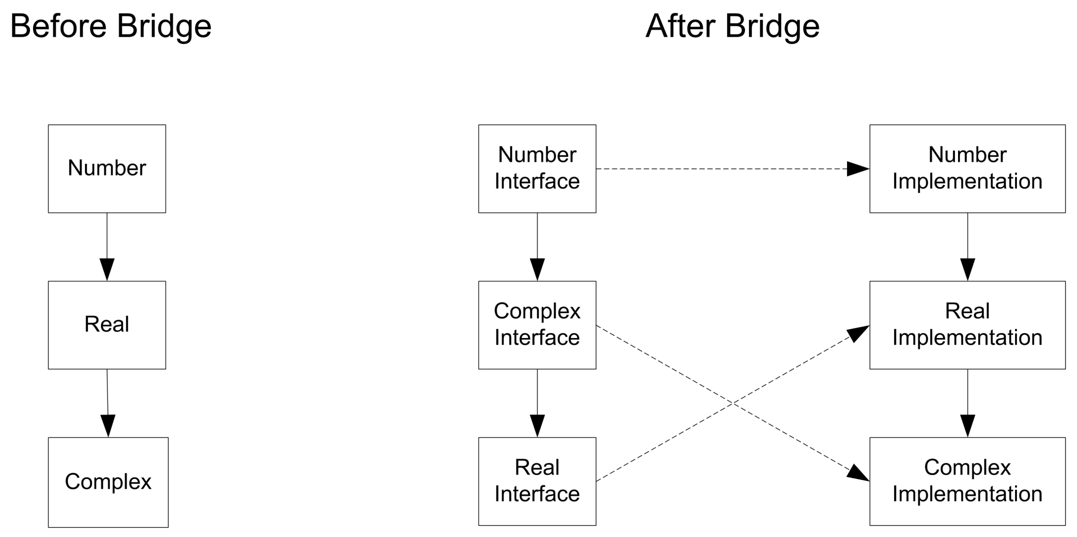

**Figure 4-3: on the left, a number hierarchy that does not implement Bridge; on the right, the original hierarchy has been split into interface and implementation hierarchies to implement Bridge. A solid arrow from Class A to Class B indicates that B inherits from A; a dotted arrow from Class A to Class B indicates that A maintains a pointer to B.**

interface. It suggests splitting the existing inheritance hierarchy into two hierarchies, one of which represents interface inheritance and the other implementation inheritance. Having each object from the interface hierarchy maintain a pointer to an object from the implementation hierarchy, whose particular type can be determined at run-time, connects the hierarchies. This is illustrated in Figure 4-3.

How does Bridge break symmetry? The inheritance transformation of Complex to Real is symmetric; a Real number can be used anywhere a Complex number is used without changing the behavior of the program. Program behavior is the invariant, and what changes is the particular subtype of the object used. Yet the symmetric solution is not good enough in some situations. By splitting the inheritance hierarchy in two – by breaking the symmetry – Bridge preserves the inheritance symmetry of the original transformation within each hierarchy, but addresses the problems of the original solution.

Bridge redistributes the existing behavioral symmetry by splitting the inheritance hierarchy into two hierarchies, each of which caters for a part of the inheritance covered in the original hierarchy. The behavioral symmetry still holds – a Real number can be used wherever a Complex number is used without changing program behavior – but the symmetry is now expressed across two hierarchies connected by reference.

### 4.3.2 Half-Object Plus Protocol (HOPP)

HOPP (Meszaros, 1995) is discussed in Section 2.2.1 and illustrated in Figure 2-1 on page 20. In a distributed systems context, HOPP transforms a single object into two “half objects plus protocol”, allowing a single object to be split across different address spaces.

Applying HOPP transforms a system in a way that breaks symmetry. Prior to HOPP, where the given concept lives in exactly one address space, the relationship between the corresponding object and each of its client objects is the same. Each of the client objects “sees” the given object in the same way, because all of the object’s functionality is in the same place. This sameness of relationship between object and client objects is a symmetry; the invariant is the relationship between object and client, and what changes is the particular client involved. This symmetry might be lost when a concept is split across object spaces, but HOPP preserves it, by breaking (or redistributing) symmetry inside the HOPP object. When forces require a concept to be implemented across multiple address spaces, HOPP thus indicates how to change the system to allow the distribution, yet still preserve the symmetry of the original system as much as possible. HOPP provides an implementation that preserves a conceptual symmetry of relationship between object and client, by redistributing symmetry in the system implementation.

This symmetry breaking increases the system’s stability. Consider a distributed system in which the HOPP pattern is present. If the system needed to expand from, say, two to three sites, it could do so easily by transforming the existing HOPP structure from two half-calls to three, and so on for further adaptation. The relationship between client and existing “half-objects” does not need to change, and the new client-object relationship is the same as that for existing client-object pairs. Coplien (1998a) cites Warren Montgomery, who notes the importance of preserving symmetry in the context of development of telecommunications software:

> Making the model symmetric, if you can, leads to great simplifications. [A system I worked on] started out symmetric, no notion of originating and terminating terminal process, just terminal processes. The call protocols were all symmetric. This made it very easy to take the next step of going to multi-party calls …
>
> (Coplien, 1998a)

### 4.3.3 Establishing a Formalism for Symmetry in Software

Coplien and Zhao (2001) have attempted to apply symmetry formalisms to the software domain. They propose a formalism for software patterns that is grounded in group theory and concepts of symmetry, and argue that several common language features can be formally classified as symmetries because they meet the mathematical requirements for such a definition.

For example, using Rosen's (1995) definition of symmetry as immunity to possible change, Coplien and Zhao argue that the interface of a class remains the same for all objects of the class, establishing an invariance relationship between the class and its objects. The class interface is thus immune to possible change with respect to the objects of the class, and it defines a symmetry. When used for subtyping, inheritance also defines a symmetry because there are behavioral invariants that hold when an object of a class is replaced by a subclass object. Overloaded operators express symmetry because they behave in the same way regardless of the type of their operands.

The symmetry of the inheritance transformation when it is used for subtyping can be described by the Liskov Substitution Principle (LSP), which can be summarized as follows:

> … there are invariants that hold (in this case, behavioral invariants) for a program under a transformation that substitutes objects of type S for those of type T. Behavior is symmetric with respect to subtyping – only here, the constancy of behavior is itself the litmus test by which subtyping is judged, not vice versa.
>
> (Coplien and Zhao, 2001)

The invariant in the symmetry is the behavior of a particular software program; what changes is the particular type of object in the program. More specifically, within any given program, there are behavioral invariants that hold when an object of type T is replaced by an object of type S, where S is a subtype of T (a distinct class derived from T). Formally, Coplien and Zhao (2001) point out that "All the invertible inheritance operations on the classes through a single subtyping path form a symmetry group for class invariants". The inheritance transformation satisfies the group theory axioms when used for subtyping: closure holds because consecutively applying the inheritance operation to a class results in a new class in the same type hierarchy as the original class; closure can be used to prove associativity; the null inheritance operation is an identity transformation; and language features such as type casting and class slicing in C++ provide an inverse.

Coplien and Zhao go on to analyze how programmers break symmetry when language constructs that express symmetry fail to solve design problems. They argue that many existing software patterns represent recurring solutions to design problems that break symmetry. The Adapter pattern, for example, breaks the type symmetry that subtyping attempts to preserve by converting the interface of one class into the interface the client expects. The Visitor pattern breaks the symmetry expressed in inheritance by creating two distinct but linked inheritance hierarchies and using successive function calls to ensure that each hierarchy plays a role in determining function invocation. In effect, the pattern provides multiple dispatch which means that a class behaves differently for different clients.

In later work, Zhao and Coplien (2002) provide more detailed analysis of symmetry in class and type hierarchy in terms of group theory and symmetry groups. Their work is, however, ongoing and represents an initial attempt to analyze symmetry in software. Their theories are only beginning to be closely evaluated. For this reason, work in this thesis is based on an informal analysis of concepts of symmetry in software, and validated by providing multidisciplinary examples.

## 4.4 Symmetry and the Theory of Centers

Alexander's earlier work (1977; 1979) was foundational in developing the concept of a pattern language as a means to articulate the process and structure by which systems evolve. Alexander developed a pattern language for architecture that identified recurring, sustaining structure in architecture and could be used to generate such structure. Eventually, however, Alexander found that pattern languages, on their own, are not sufficient to generate good structure (Coplien, 1998a).

Alexander's (2002a; 2002b) more recent work has been to develop an understanding of order that goes beyond architecture to include biology and physics. In Alexander's context, good structure means beautiful buildings, and his quest to build beautiful buildings led to his more recent work. This work is more broadly focused than his earlier work and provides a context for using pattern languages as part of a process of generating beautiful structure. Alexander argues that beauty is not about image, nor is it opposed to functionality, but rather beauty and functionality are one and the same when understood correctly. Both are inherently tied to structure, which leads Alexander to address two fundamental questions: "What is good structure?" and "How can humans generate good structure?"

While Alexander's primary interest is in architecture, the scope and relevance of his work are much broader. In Alexander's terms, good structure is that which makes something whole; he thus calls it "wholeness". Wholeness arises out of many parts working together to create something that is greater than their sum. Surprisingly, Alexander has found that wholeness is also objective in the sense that it is consistently recognized as such, by a wide range of people, and can thus be considered universal. In Alexander's terms, wholeness is identifiable in terms of the presence and strength of *centers* (Alexander's term for fundamental building blocks) in a system. A center is a coherent, structural entity that is part of a larger whole.

Alexander identifies fifteen key properties that help to build and strengthen centers. He argues that the degree to which these properties are present in a system is a measure of the strength of the system's centers and thus the wholeness of its structure. Table 1 lists Alexander's properties in order of importance.

There are many connections between symmetry and Alexander's fifteen fundamental properties. For example, property seven identifies the importance of local symmetries, and the properties of good shape, deep interlock and ambiguity, contrast, and echoes are all created, in part, by symmetries. Alexander himself notes the importance of symmetry to his work:

> The wholeness is an autonomous and global structure, which is induced by the details of the configuration. It is a real physical and mathematical structure in space – but it is created indirectly, by symmetries and other relationships which are induced in the geometry.
>
> (Alexander, 2002a, p. 86)

While the role of symmetry breaking in patterns is widely recognized, at least in physics, little work has been done in the area of pattern languages and symmetry breaking. Yet, if patterns are transformations on system structure that break symmetry, and patterns in a pattern language work together to generate a system, then pattern languages must be able to be analyzed in terms of symmetry breaking. Such analysis is a significant contribution of this thesis, and is outlined in more detail in Chapter 5 and Chapter 6.

| **Property (in order of importance)** | **Description of Property** |
|---|---|
| 1. Levels of Scale | Centers need to be of a varying size, with definite but small jumps between sizes. |
| 2. Strong Centers | Individual centers should be intensified by their position in the whole. |
| 3. Boundaries | A center ought to be surrounded by a boundary whose size is of the same order of magnitude as the center itself. The purpose of a boundary is twofold. First, it draws attention to the center and thus helps produce it. Second, it joins the center with the world beyond the boundary, by providing a common place between the two. |
| 4. Alternating Repetition | Centers intensify other centers by their repetition. The repetition must not be so simple that it is boring, but must create a second system of centers that complements the first. Alexander likens the effect to that of counterpoint in music. |
| 5. Positive Space | Positive space occurs when there is no space that is just "leftover", but, rather, each piece of space is substantial in itself. |
| 6. Good Shape | The quality that makes shapes beautiful is hard to define, but is tied to centers. Good shape is made up of multiple, coherent centers. It has a recursive aspect: the elements of a good shape must also be good shapes themselves. Good shape can be built from elementary, regular shapes, such as diamonds, triangles and squares, as opposed to amorphous blobs. |
| 7. Local Symmetries | The symmetries that are important to wholeness are those that hold within limited pieces of a design. They leave the whole flexible and able to adapt: "*overall* symmetry in a system, by itself, is not a strong source of life or wholeness" (Alexander, 2002a, p. 186). |
| 8. Deep Interlock and Ambiguity | A strong center tends to be "hooked" into its surroundings, so that it is difficult to disentangle the two. |
| 9. Contrast | Differentiation is central to creating strong centers; unity can only be created out of diversity, out of distinctness. |
| 10. Gradients | Qualities vary subtly and gradually across space. |
| 11. Roughness | Roughness is not "a residue of technically inferior culture, or the result of handcraft or inaccuracy. It is an essential structure feature … without which a thing cannot be whole." (Alexander, 2002a, p. 68) Designs rarely fit exactly into their space, requiring shapes to be drawn imprecisely. That morphological inexactitude is critical to good design, creating a better design than one that fits exactly into its space. |
| 12. Echoes | Individual elements have deep, internal structural similarities, recognizable as a family-like resemblance. |
| 13. The Void | A center often needs deep emptiness at its heart. Too much detail is self-destructive. |
| 14. Simplicity and Inner Calm | Everything that is unnecessary should be removed. |
| 15. Not-Separateness | A center must be connected to its surroundings, and even unable to be completely separated from them. |

**Table 1: The fifteen fundamental properties described by Alexander.**

Alexander argues that the only way that good structure can be generated is by an *unfolding* process. Such a process is one in which growth is incremental and always seeks to benefit the whole, in which there is continuous feedback about possible directions for growth, and in which the unpredictable nature of that growth is accepted and valued.

Brand (1997) affirms the importance of an unfolding process in designing good buildings. He argues that slow, small-step growth – piecemeal growth – is what makes a building grow gradually better rather than gradually worse over time. Piecemeal growth is not just healthy for individual buildings. Brand argues that much wholesome evolution of cities can be explained by the persistence of site, by which he means that property lines and thoroughfares in cities do not change even when, for example, hills are leveled and waterfronts filled in. He (1997, pp. 156-177) notes how important it is for buildings to be able to adapt over time, according to feedback from their users, and that vernacular design does just this, and is therefore most effective. Where vernacular design is not possible, he recommends a building-style called scenario planning as an approach to architecture that encourages building users to design buildings that will cope in many possible future situations, rather than focusing entirely on present needs.

In Alexander's view, pattern languages are necessary to the development by humans of good structure because they provide contextual, cultural information necessary to strengthen centers. He further argues that if a pattern language is to be capable of generating good structure, his fifteen properties should be present in the language to a high degree, and the language should embody an unfolding process of design.

# Chapter 5

# A Language Designer's Pattern Language

This chapter focuses on how to help pattern language writers build better pattern languages. It presents a pattern language called the Language Designer's Pattern Language (LDPL) that itself articulates the structure of pattern languages and key processes by which they form and evolve, and thus guides the building of pattern languages. The focus of LDPL is semantic, rather than syntactic; it focuses on how patterns work together to build a system. The theoretical foundation of LDPL is Alexander's (1977; 1979) work on pattern languages and Stewart and Golubitsky's (1992) work on symmetry breaking as a means of analyzing complex systems.

Section 5.1 outlines the key assumptions behind the development of LDPL. Section 5.2 outlines key insights from design theorists relevant to the development of LDPL and Section 5.3 provides an overview of examples used in LDPL. The pattern language itself is described in Section 5.4.

## 5.1 A Basis for LDPL

The main assumptions underlying the development of LDPL are as follows:

- It is valid to treat a pattern language as a designed system, and therefore valid to develop a pattern language for building pattern languages, based on theory about system design and evolution.
- A pattern language is both a complex system and a means of analyzing complex systems.

- Natural, physical systems provide insight useful for building and maintaining designed systems.

### 5.1.1 A Pattern Language is a Designed System

In a discussion example related to garden design, Alexander points out that a pattern language is, in fact, a designed system:

> … the design of the garden lies within the language for the garden …
>
> We tend to imagine that the design of a building or a garden takes a long time; and that the preparation for the process of design is short. But in the process where the language plays its proper part, this gets reversed …
>
> Essentially, this means that the language which you have prepared must be judged as if it were itself a finished garden (or a finished building) …
>
> … if you think of the language merely as a convenient tool, and imagine that the garden of the building you create will become beautiful later, because of the finesse with which you handle it, but that the collection of patterns which lie in the language now are not enough to make it beautiful, then there is something deeply wrong with the language …
>
> 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.
>
> (Alexander, 1979, pp. 323-324)

A consequence of being a designed system is that a pattern language is structural. It is not simply a guide to some kind of process of design. It is a structural entity that articulates design structure as well as design process, and is more powerful than an isolated pattern. The structure of a pattern language comes both from the structure of individual patterns and from their interaction; the patterns in a language work together to generate a system.

### 5.1.2 A Pattern Language is a Complex System

A complex system is "a system formed out of many components whose behavior is emergent, that is, the behavior of the system cannot be simply inferred from the behavior of its components" (Bar-Yam, 1997, p. 10). Governments, families, and the human body are all examples of complex systems, whereas a pendulum, spinning wheel, and orbiting planet are all simple systems. The amount of information needed to describe a complex system provides a measure of its complexity. Understanding a complex system requires understanding its individual parts, their interdependencies, and how those parts act together to form the behavior of the whole system. Complex systems may have been designed, or have evolved naturally, or have partly evolved and partly been designed. Systems in domains as diverse as nature, software, economics, and ecology can thus be classified as complex systems.

According to Alexander's (1979) work, a pattern language fits the above definition of a complex system, because it is made up of components and its overall behavior is emergent; the behavior of a pattern language cannot be explained simply by studying individual patterns:

> The structure of a pattern language is created by the fact that individual patterns are not isolated.
>
> …
>
> Each pattern then, depends both on the smaller patterns it contains, and on the larger patterns within which it is contained.
>
> (Alexander, 1979, p. 309-313)

> [Patterns] … are complex, and potent fields.
>
> Each pattern is a field - not fixed, but a bundle of relationships, capable of being different each time that it occurs, yet deep enough to bestow life wherever it occurs.
>
> A collection of these deep patterns, each one a fluid field, capable of being combined, and overlapping in entirely unpredictable ways, … [is] capable of generating an entirely unpredictable system of new and unforeseen relationships.
>
> (Alexander, 1979, p. 223)

In reviewing Alexander's work, Salingaros reiterates the emergent nature of pattern languages:

> Not only does each original pattern work in combination as well as it did individually, but the whole contains organizational information that is not present in any of its constituent patterns.
>
> (Salingaros, 2000)

Bar-Yam (1997, pp. 1-2) notes that the field of complex systems research regards all complex systems, regardless of their particular domain, as having universal properties. Insights from one domain can therefore be used in another domain by recognizing their universal applicability, as happens in the Theory of Inventive Problem Solving methodology (TRIZ in its Russian abbreviation). TRIZ grew out of the work of Russian scientist Genrikh Altshuller (1984), who studied the creativity of the inventor by analyzing hundreds of thousands of invention descriptions from world-wide patent databases. Altshuller identified key, recurring strategies facilitating the solution of complex, technical problems across a wide variety of domains. He went on to argue that all technological systems, regardless of their particular domain, evolve according to patterns that are objective and predictable. Recognizing those patterns is key to releasing the creativity needed to solve problems in complex systems. Senge (1990, p. 73) points out that the key to understanding complex systems is to understand their dynamic complexity; one should not fight complexity with complexity, but rather learn to "recognize types of 'structure' that recur again and again". Dooley (1997) makes the same point when he says that complex systems can be understood by "looking for patterns within their complexity, patterns that describe potential evolutions of the system". Salingaros (2000) notes that a pattern language can be used to understand a complex system because it both identifies the patterns of system evolution underlying the system, and shows how those patterns work together to create emergent behavior. Alexander points out that patterns in a pattern language describe potential system evolutions:

> A pattern language is a system which allows its users to create an infinite variety of those three dimensional combinations of patterns which we call buildings, gardens, towns.
>
> First, it defines the limited number of arrangements of spaces that make sense in any given culture. This is a far smaller collection than the total number of arrangements of jumbled nonsense, the *piles* of bricks and space and air and windows, kitchens on top of freeway interchanges, trees growing upside down inside a railway station - that could be put together, but would make no sense at all.
>
> And second, a pattern language actually gives us the power to generate these coherent arrangements of space. Thus, 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)

### 5.1.3 Evolved Systems Provide Insight

Recognizing the common properties (Bar-Yam, 1997, pp. 1-2) in complex evolved systems and complex designed systems enables insights from evolved systems to be used to understand and build designed systems. Dasgupta (1991) argues that it is valid and worthwhile to view design, and in particular design in computer science, as an evolutionary process. While deliberate design is clearly different from the Darwinian view of natural evolution, which involves a process of variation and natural selection, key aspects of solution development related to testing and adaptation are the same for both processes. In particular, solving complex problems typically requires making choices involving many alternative and mutually interacting factors. Design solutions to complex problems are therefore satisficed; they are good solutions, suboptimal but workable, rather than the best solution. Indeed, sometimes the optimal solution cannot be comprehended, and even if it can, it may not be practical to use it. Many models suggest that design is a process of iterative refinement, as is evolution. Any design solution should therefore be viewed as tentative and likely to evolve further.

In terms of the processes by which design takes place, Dasgupta (1991, pp. 114-115) describes both ontogenic and phylogenic design. Ontogenic design is an "unfolding and development of a form from some initial state that is at odds with the requirements to some state that fits the requirements", and phylogenic design is an "act of design [that] involves *successive changes or improvements to previously implemented designs*". He argues that both ontogenic and phylogenic design can be viewed as evolutionary processes. Further, the phylogenic design process appears to follow the same evolutionary model that explains ontogenic design, with one cycle of phylogenic design usually constituting one or more cycles of ontogenic design evolution. Dasgupta analyzes the kind of typical design processes within each of four typical design paradigms in computer science and argues that the design processes carried out within those paradigms can be classified as either ontogenic or phylogenic and can therefore be viewed as evolutionary design.

## 5.2 Key Insights from Design Theorists

Rather than focus directly on system design, LDPL aims to help designers create the conditions necessary for good design to happen. In a sense, then, LDPL is about meta-design, which can be defined as follows:

> [Meta-design is about creating] the enabling conditions for design activities. In other words [it provides] a perspective of design in which the user is a designer.
>
> An important aspect of meta-design is to design not just an artifact, but a life-cycle that anticipates changes that may occur over a long period of time.
>
> (Ostwald, 2002)

As Coplien (2000b, p. 262) points out, in traditional and contemporary software design, the approach has usually been to select a tool from a particular paradigm and fit a design to that paradigm. A better approach is to determine what kind of paradigm is the most appropriate for a particular design, and then choose a tool that fits that paradigm.

Dasgupta (1991, p. 141) points out that the sense in which the concept of paradigm is usually used in design domains like computer science tends to be simplistic, but can still be understood within a broader Kuhnian framework: "*A design paradigm is to design what a set of heuristic and/or metaphysical models (within a Kuhnian paradigm) is to science.*" A Kuhnian paradigm represents a highly sophisticated understanding of paradigm, and represents the fundamental viewpoint from which a scientific community sees the world. Dasgupta notes that Kuhn uses two related concepts to provide a model for his notion of paradigm: a disciplinary matrix, and exemplars. The matrix articulates fundamental concepts underpinning the paradigm: assertions that the discipline community assumes to be true, beliefs in abstract or physical models to which the domain is assumed to conform, and values held by the discipline community. The exemplars are shared sample problem/solution pairs that characterize the discipline. Understanding the exemplars is thus key to designing within the discipline.

Using Kuhn's notion of paradigm, Dasgupta characterizes five typical design paradigms in computer science: Analysis-Synthesis-Evaluation (ASE), artificial intelligence, algorithmic, formal design, and the Theory of Plausible Designs (TPD). As its name suggests, the ASE paradigm involves successive stages of first analysis, then synthesis, then evaluation of the design against requirements. The artificial intelligence paradigm treats problems as ill-structured. Design in this paradigm involves describing the problem in terms of initial, intermediary, and goal states. Operators allow transitions from one state to another. Searching for the solution to a problem involves applying a sequence of operators to move through the problem space. The algorithmic paradigm assumes that problems are well-structured and the problem can be solved with a domain-specific algorithm. The formal design paradigm treats a design as theorem that can be proved to solve a problem. In contrast, TPD views the design process as continually evolving and only ever correct at a given point in time and space.

Winograd and Flores (1986) explore design theory with particular reference to the design of computer technology. They first explicitly outline the philosophical, language, and thinking traditions or patterns that implicitly underlie much of Western thought. They go on to explore how these usually assumed and implicit cultural traditions influence the kind and effectiveness of the computer technology we develop. The conclusion they draw is that some of these patterns/traditions do not help us build "good" computer technology, because the assumptions on which they are based are outdated, inaccurate or inappropriate for the area concerned. They explore other philosophical and language traditions that present different and perhaps more correct assumptions, and they suggest future directions for the design of computer technology. One of the key points that Winograd and Flores make is that successful design is not about anticipating every possible change. Rather, successful

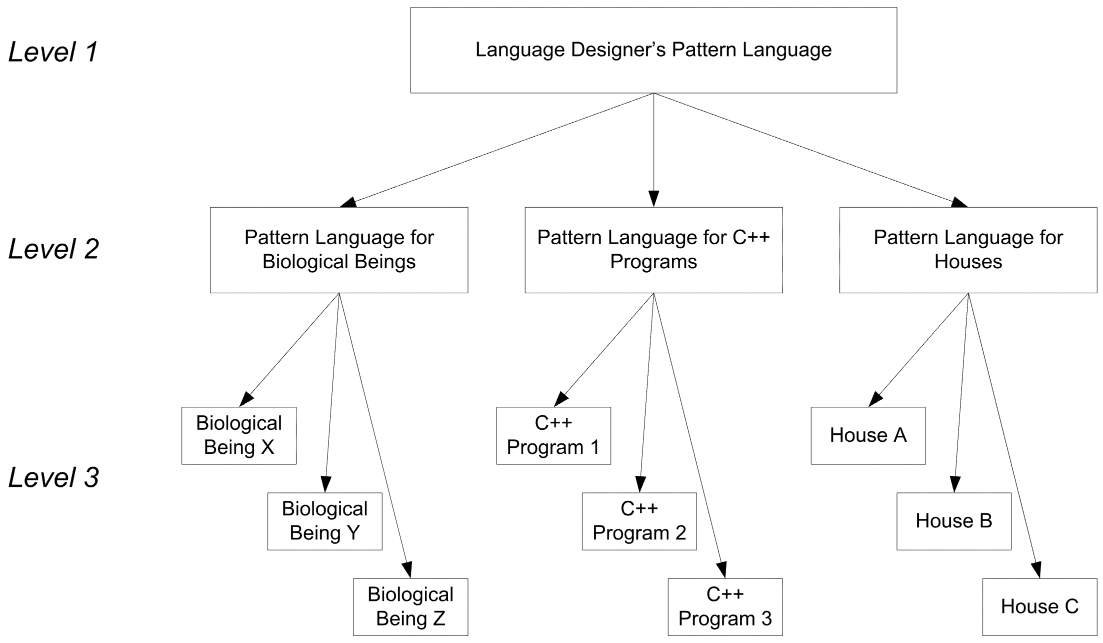

**Figure 5-1: Language Designer's Pattern Language (LDPL) helps build pattern languages for system design, that in turn help build physical systems.**

design requires identifying the fundamental structure of a domain and recognizing that the structure governs the kind of change likely to occur. A successful design is therefore one that can evolve within that structure:

> The most successful designs are not those that try to fully model the domain in which they operate, but those that are "in alignment" with the fundamental structure of that domain, and that allow for modification and evolution to generate new structural coupling. As observers (and programmers), we want to understand to the best of our ability just what the relevant domain of action is. This understanding guides our design and selection of structural changes, but need not (and in fact cannot) be embodied in the form of the mechanism.
>
> (Winograd and Flores, 1986, p.53)

## 5.3 Examples Used in LDPL

LDPL is intended for use in the development of *pattern languages*. Those pattern languages can then in turn be used in the development of *physical systems*, as shown in Figure 4-3. For example, LDPL could be used to help build a pattern language for C++ programs, which could then be used to help develop C++ programs.

The existence of each pattern in LDPL is confirmed by using examples from existing pattern languages for system development – languages at the second level of the diagram shown in Figure 5-1. The examples are drawn from several different languages in several different application domains, substantiating the multidisciplinary nature of LDPL and validating that the symmetry-breaking theory underlying LDPL is, in fact, fundamental. This validation is particularly important in the given context, in which formal analysis of symmetry-breaking concepts in many domains has not yet been fully developed.

Software is one domain from which pattern language examples are drawn, because it is the domain of interest for this thesis. Architecture is another, because of architect Christopher Alexander's (1977; 1979) foundational work on pattern languages and their subsequent influence on the development of software patterns discipline.

Each pattern in LDPL also includes examples drawn from biological systems – entities at the third level of the diagram shown in Figure 5-1. Biological examples are of particular interest because of the existence of formal analysis of biological systems in terms of the symmetry and symmetry breaking theory (Stewart and Golubitsky, 1992) on which LDPL is based. Ideally, any biological examples would be drawn from biological pattern languages, rather than biological systems, but such languages have not yet been explicitly articulated. The biological examples have therefore been carefully chosen to reflect structures likely to be present in a biological pattern language.

The multidisciplinary examples included with each pattern in LDPL come from a variety of sources. From the domain of software patterns and pattern languages, Coplien's (2000a) C++ Idioms Pattern Language, the CHECKS Pattern Language of Information Integrity (Cunningham, 1995) and the Half Object Plus Protocol (HOPP) pattern (Meszaros, 1995), are used extensively as examples. Alexander's (1977) architectural patterns provide a rich source of architectural pattern language examples. The biological examples relate to diverse processes and structures, including the processes by which cells develop and regenerate, and the social structures of insect communities.

The C++ idioms language (Coplien, 2000a) draws on idioms from earlier work by Coplien (1992) to develop a pattern language that guides the building of an inheritance hierarchy in C++, with particular focus on algebraic types. Key issues in building such a hierarchy include efficient and effective memory management, placement of algebraic operations in the hierarchy, and implementation of operations that depend equally on more than one operand. The idioms language is summarized in Section 2.4.1.

The CHECKS pattern language (Cunningham, 1995) guides the development of software that accepts user input. The language shows how to separate good input from bad and how to minimize the amount of bad input recorded. It consists of ten patterns relating to quantifying the domain model, providing feedback about the domain model to the user as transparently as possible, and addressing the long-term integrity of information. CHECKS is summarized in Section 2.4.1.

The HOPP pattern (Meszaros, 1995), discussed in Sections 2.2.1 and 4.3.2, describes how to decompose a concept that exists across multiple address spaces. It splits an original, whole object into two half-objects plus a communication protocol between them. For the examples in LDPL, HOPP is discussed in context with how it interacts with various other patterns, to provide language-like examples, rather than examples focused on one pattern.

Alexander's (1977) architectural pattern language for towns, buildings and construction provides "a coherent picture of an entire region", from large-scale regional design down to the details of construction of particular buildings. The patterns can be combined in many different ways to generate a variety of structures.

## 5.4 The Pattern Language

LDPL addresses the following key issues:

- generating patterns from a system of forces
- identifying and understanding the nature of relationships between patterns
- adding new patterns to, and removing patterns from a pattern language
- maintaining stability in a pattern language over time

These issues are important because they are key issues from the point of view of building pattern languages for complex systems, where those systems need to function effectively, over a long term, in a dynamic environment. They relate both to how patterns work together, and to the structural properties of systems generated with pattern languages built from this pattern language.

LDPL identifies ten patterns, as shown in the language structure diagram in Figure 5-2. *Local Repair* is the pattern largest in scale, and it highlights the importance of structural transformations being small-step and local in scope, but in accord with

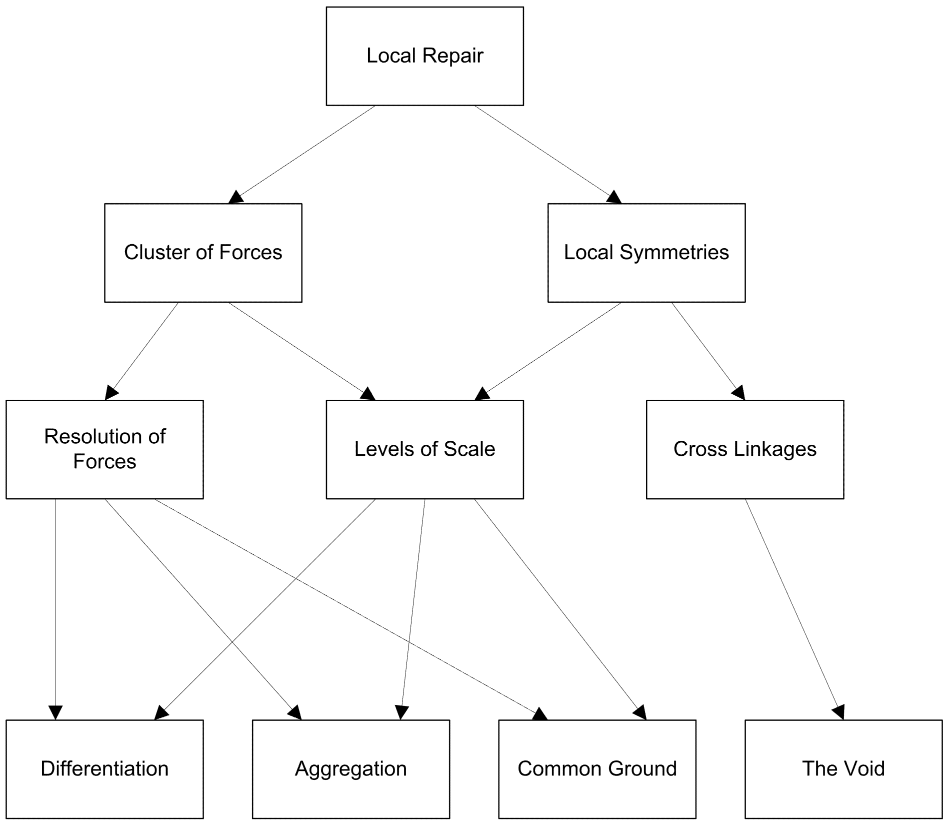

**Figure 5-2: A diagram showing the language structure of LDPL. The higher a pattern in the diagram, the larger its scale. An arrow from pattern A to pattern B indicates that B builds on A.**

global needs. *Cluster of Forces* focuses on how to identify and group together related forces, and *Resolution of Forces* focuses on balancing identified groups of forces to create patterns. *Local Symmetries* shows how to connect patterns in a language and *Levels of Scale* ensures that connected patterns are relatively close in scale. *Cross Linkages* addresses the problem of the language structure being rich enough to generate complex system structure. *Differentiation, Aggregation,* and *Common Ground* show different ways patterns are added to a language. *Differentiation* shows how to elaborate an existing pattern when finer structure is needed, *Aggregation* shows how to incorporate structure that appears new and different, and *Common Ground* points out that a new pattern may arise out of existing patterns in different contexts with underlying commonality. *The Void* focuses on removing a pattern or patterns from a language.

## Local Repair

*If you have an isolated or unbalanced force*
*Then try to accomodate it in an existing pattern.*

### *Context:*

You have a pattern language that at one time has balanced its forces, but now you have a force that is isolated or disturbs the balance of the language.

### *Problem:*

You need to incorporate the force into the language in a way that minimizes the changes required and benefits rather than detracts from the whole language.

### *Forces:*

- The information available about forces will never be complete.
- The process of finding a solution that balances some forces will cause other forces to arise; it is not possible to enumerate all the forces before attempting a solution (Grabow, 1983, pp. 40-41).
- The world is always changing, so new forces are always coming and going from the design (Grabow, 1983, pp. 40-41).
- No force can be considered in isolation. Forces are interrelated, and a given pattern resolves a given set of forces:

  > In short, a pattern lives when it allows its own internal forces to resolve themselves.
  >
  > …
  >
  > The "bad" patterns are unable to contain the forces which occur in them.
  >
  > (Alexander, 1979, pp. 120, 129)

- A collection of patterns works together to resolve a system-wide set of forces:

  > The individual configuration of any one pattern requires other patterns to keep itself alive.
  >
  > …
  >
  > It is true that each [pattern] can be explained, in its own terms, as an isolated thing, which is needed to resolve certain forces. But, also, these few patterns form a whole, they work as a system ….
  >
  > (Alexander, 1979, pp. 131, 133)

> [The] life of [a] pattern … [depends on] the self-sustaining harmony, in which each process helps sustain the other processes, and in which the whole system of forces and processes keeps itself going, over and again, without creating extra forces that will tear it down.
>
> (Alexander, 1979, p. 118)

- The need for local repair to a language is driven by the need for repair in systems generated by the language.
- Re-clustering a set of forces from scratch could lead to instability in systems generated with the language, because the patterns in the new language would not have been validated by their repeated occurrence in existing systems.
- Focus is by nature local; a designer can only be attentive to one thing in one place at a time (Raskin, 2000, pp. 9-32).

### *Solution:*

Identify the pattern that can best accommodate the isolated or unbalanced force (where the force best fits in) and focus on that pattern. Often, the new force is implicit in an existing solution and simply needs to be explicitly stated; the force can then be added to the pattern without disturbing the balance of the forces. When the force is truly new, the old solution can often be adapted to re-balance the forces. Even if it turns out that the new force requires a new solution, the old solution will often serve as a solid basis for the new solution.

Doing local repair on a language might require adding a pattern, removing a pattern, or modifying a pattern. As a result of these changes, the interconnections between patterns may also need to be modified.

### *Resulting Context:*

The overall system of forces remains balanced, despite possibly having changed, and the patterns in the language still fit together appropriately. If, however, many forces are missing from the language as a whole, then incorporating a newly identified force into an existing pattern may reinforce poor structure rather than complement good structure.

### *Rationale:*

Alexander (2002b, pp. 203-228) describes what he calls the Fundamental Process, which he argues is the key process governing the development of self-sustaining systems. The fundamental process drives change through local and small-step modification that has a global focus. Change happens by evaluating the system to see which small-scale, local change would best impact the whole. That local change is made, and its effect on the whole evaluated. If the effect on the whole is negative, the change is wiped out, and the process begins again.

Awareness of the impact of local change on a system as a whole is critical because in an interconnected system, change to one part of a system inevitably affects other parts of the system. Alexander argues that the fundamental process helps build stable systems because change is based on what has worked in the past rather than what is anticipated to happen in the future, and because it is incremental and thus gives an opportunity for feedback, with the effects of small changes able to be studied before other changes are made, and altered if necessary. Further, such incremental change implies a willingness to live with unpredictability - to resist defining *a priori* the nature of major change, but rather let that change unfold step by step.

Stewart Brand (1997) argues that slow, small step growth is what makes a building grow gradually better rather than gradually worse over time. Such growth is not just healthy for individual buildings. Brand argues that much wholesome evolution of cities can be explained by the persistence of site: in other words, by the fact that property lines and thoroughfares in cities do not change even when, for example, hills are leveled and waterfronts filled in. Brand also points out that most building design is vernacular and vernacular design is inherently small step and localized.

Senge (1990) emphasizes the importance of local but globally aware change in his study of complex systems. He argues that when making changes to a system, global awareness is critical, but it is not the scale on which most changes should be made. Dynamic complexity must be managed locally (Senge, 1990, pp. 290-301), but local management of dynamic complexity must be driven by global, as well as local concerns. Senge points out the typical cycle that happens in many complex systems when local repair is driven by local concerns, and the needs of the whole are ignored:

> In the short run, individuals gain by acting selfishly. This selfishness leads to success, which reinforces the actions that led to the success. … But, the sum total of all the individuals acting in their self-interest … add up to a "total activity" with a life of its own. Eventually …, the unsustainable "gain per individual activity" … begins to decline, and the benefit to each individual begins to reverse. By the time they realize the import of that common error, it's too late to save the whole, and all the individuals fall with it. It's not enough for one individual to see the problem; the problem cannot be solved until most decision makers act together for the good of the whole.

> (Senge, 1990, pp. 295-296)

Alexander uses a biological analogy to make the same point:

> Each part (cell) is free to adapt locally to its own processes, and is helped in this process by the genetic code which guides its growth.
>
> Yet at the same time, this same code contains features which guarantee that the slow adaptation of the individual parts is not merely anarchic, and individual, but that each part simultaneously helps to create those larger parts, systems, and patterns which are needed for the whole.
>
> (Alexander, 1979, p. 165)

The structure of a pattern language is that each pattern is smaller in scale than those patterns above it, and each pattern, by definition, takes into account the structure of those patterns above it. This structure inherently takes into account global context and ensures that implementing change by focusing on one pattern incorporates global awareness:

> At any given moment in the unfolding of a sequence of patterns, we have a partly defined whole, which has the structure given to it by the patterns that come earlier in the sequence.
>
> (Alexander, 1979, p. 390)

> At every level, certain broad patterns get laid down: and the details are squeezed into position to conform to the structure of these broader patterns. Of course, under these circumstances, the details are always slightly different, since they get distorted as they are squeezed into the larger structure already laid down. In a design of this type, one naturally senses that the global patterns are more important than the details, because they dominate the design. Each pattern is given the importance and control over the whole which it deserves in the hierarchy of patterns.
>
> (Alexander, 1979, pp. 381-384)

> Each pattern encompasses, and contains, the forces which it has to deal with; and there are no other forces in the system. Under these circumstances, each event which happens is resolved. The forces come into play, and resolve themselves, within the patterns as they are.
> Each pattern helps to sustain other patterns.
>
> (Alexander, 1979)

> The language will evolve, because it can evolve piecemeal, one pattern at a time.
>
> (Alexander, 1979, p. 344)

### *Example (architectural pattern language):*

While designing a campus for the University of Oregon (1975, pp.101-143), Alexander and colleagues identified some patterns from *A Pattern Language*

(Alexander *et al.*, 1977) for possible use in the project. One of the patterns identified was Building Complex (1977, pp. 468-472), which says that a monolithic building structure is unhealthy, because it tends to depersonalize the communication between users of the building and staff in the building, whereas a building structure that is more differentiated encourages users of the building to identify with the staff there as personalities, rather than just personnel.

Given that a university needs some kind of central administration, it would seem sensible to apply Building Complex to the building of a large centralized campus to house administration offices for the university. Users of the campus, however, realized that while a university does need some kind of centralized administration, housing it in a single, large building complex is ineffective, because it places some administrative services too far away from the functional connections they need in the community. This represented a new force that needed to be incorporated in the language for the project. Rather than throw away Building Complex, Local Administration suggests locating parts of the administration in different buildings around campus, such that particular administrative services are located as closely as possible to the particular communities they serve. For example, faculty-specific administrative staff might be located in the building of the particular faculty they serve. Administration thus might logically function as a centralized unit, but there is some decentralization of staff location. The buildings still form a building complex, but that complex is now distributed around a university campus. Modifying the language by recognizing that the new force relates to Building Complex, and incorporating a new pattern that builds on Building Complex is an example of *Local Repair*.

### *Example (C++ Idioms Pattern Language):*

One force missing from the idioms language (Coplien, 2000a) is that C++ allows interface to be separated from implementation using pointers, but this is not effective because it makes it impossible to overload built-in operators. This new force clearly fits best with the Handle/Body pattern, whose other forces also relate to separating interface and implementation. The solution structure of the existing Handle/Body pattern already balances the new force – it overcomes the problems of using pointers to separate interface and implementation – so all that is needed is for the force to be explicitly stated; this is the simplest kind of *Local Repair*.

An early version of Promotion Ladder (Coplien, 2003) was based on a class promoting objects of derived class types to its own type. This set the scene for Homogeneous Addition to work. At some point, however, the author realized that there was a force that Promotion Ladder needed to resolve but did not. Specifically, the original version of Promotion Ladder allowed base classes to know about their children. This prevented important forces about encapsulation – for example, that a class should not know about its derived classes – from being resolved. The author recognized that the new force belonged with Promotion Ladder and adjusted the solution of Promotion Ladder to take account of the newly recognized force. The change was to make derived classes promote themselves to their base class type, rather than the other way around. This change allowed all forces to be resolved, and represents an example of *Local Repair* where existing structure is modified.

The idioms language was originally developed without Detached Counted Body, which shows how to keep track of the number of handles to a body without modifying body code. At some later point, the author realized that some systems generated with the language needed to be able to add to a Handle/Body context the capacity to manage multiple handles for one body, but without access to Body code. This realization represented a new force: that the body code could not be modified. While this force is most closely related to the forces in the existing Counted Body pattern, it cannot be potentially balanced together with those forces, because one of the forces already implicit in Counted Body and critical to its solution is that the body code can be modified. The force therefore required a new pattern; Counted Body and Detached Counted Body both represent valid solutions to a problem, each appropriate in a different context. This example of *Local Repair* resulted in existing structure being further elaborated and a new pattern being incorporated in the idioms language.

### *Example (CHECKS language):*

Whole Value (Cunningham, 1995) suggests that specialized values meaningful in the application domain should be constructed and used for message arguments and as input/output units. In the context of a document editor, where characters represent fundamental domain values, Whole Value suggests that characters should be implemented as objects. Implementing characters as objects offers several advantages (Gamma *et al.*, 1995). It allows the "application's object structure … [to] mimic the document's physical structure", which allows characters and more complex elements to be "treated uniformly with respect to how they are drawn and formatted".

However, the cost of representing characters as objects can become prohibitive, as "even a moderate-sized document may require hundreds of thousands of character objects". This prohibitive cost represents a new force, which becomes operative at the point at which cost is unmanageable. One way of incorporating the new force is to represent characters as built-in types, rather than objects. Such a solution, though, comes at a high cost because it negates the advantages of representing characters as objects. Moreover, it requires changing the system globally, and does not preserve existing symmetry.

Flyweight (Gamma *et al.*, 1995, pp. 195-206) shows how to retain the benefits of storing simple entities as objects, but defray the potentially high storage cost of doing so. Flyweight shows how to use sharing to support large numbers of simple objects efficiently. In the document editor, one Flyweight object can be created for each character in a character set and a single instance of that object shared amongst all occurrences of the character in the document. Information that differs between character instances, such as font, size, and position on a page, can be calculated "from text layout algorithms and formatting commands in effect wherever the character appears".

In the context of a pattern language incorporating Whole Value, and at the point at which the cost of representing particular entities as objects becomes unmanageable, the addition of Flyweight to the language represents *Local Repair* of the language, in the region of Whole Value.

### *Example (biological system):*

When a tree grows, it begins with a trunk and then splits into various branches. The way the branches form is for each split to form smaller and smaller branches. For example, the trunk might split into two branches each about half the size of the original trunk, then each of those halves might split into two pieces, smaller again, and so on. Whenever the leaves on a branch begin to get too heavy, this represents a new force acting on the system. The branch responds by growing a new, smaller sub-branch; the change is local and fits with the global needs of the tree in the sense that the tree doesn't collapse because of the new sub-branch that has grown.

### *Related Patterns:*

*Cluster of Forces* shows how to identify where a force belongs in the language. *Local Symmetries* shows how to connect patterns in the language together. *Levels of Scale* shows how the language as a whole is organized.

## Cluster of Forces

*If there are too many forces to think about all at once*
*Then use domain knowledge and structure to make clusters of forces.*

### *Context:*

You are developing a pattern language. A pattern language is a field of forces. The forces are numerous, interrelated and often conflicting. You have a force or forces that you need to place into the language.

### *Problem:*

You need to partition forces into patterns so that each pattern is as independent as possible, allowing patterns to be applied one at a time, in sequence.

### *Forces:*

- Where there are many interrelated forces to be considered, their interactions are too complex for the designer to grasp simultaneously (Alexander, 1964, p. 59).
- It is important to take account of how responding to one force affects other forces; responding to forces in isolation leads to inequilibrium (Alexander, 1964, p. 51).
- Forces in a pattern should not cause or reflect needless change. Otherwise, minor disturbances will take hold of the pattern and distort it (Alexander, 1964, p. 52).
- For any one force or collection of forces, there are certain other forces that must be considered simultaneously with it in order for the overall system of forces to be balanced (Alexander, 1964, pp. 1-2).
- "[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)

### *Solution:*

Group forces together so that they form clusters that maximize the interaction of forces within clusters and minimize it between clusters (Alexander, 1964, p. 124). Each cluster will form the basis for a pattern in the language. To work out which forces interact the most, use domain knowledge. A main consideration in this is to cluster those forces that relate to the same domain structure (Alexander, 1964, p. 132).

### *Resulting Context:*

Each cluster of forces will form the basis for a pattern in the language, so you can focus on each cluster individually, confident that resolving the forces in that cluster will not significantly unbalance forces in other clusters. The patterns in the language will be in alignment with the domain, representing key domain structures. If, however, many forces from the language have not yet been identified, you run the risk of having developed a poor clustering, and may need to redo the clustering from scratch at a later stage.

### *Rationale:*

Designers using a pattern language will often not fully understand all of the forces at play. The pattern language should enable the designer using the language to create well-structured entities in the domain of interest without having to fully comprehend the interrelationships of the forces they are dealing with. Maximizing the interaction of forces within patterns and minimizing the interaction between patterns enables designers to apply patterns one at a time with each pattern independently resolving a different set of forces.

Patterns in a pattern language represent subdivisions of the design problem into smaller problems whose forces are as independent as possible. This facilitates pattern language users addressing potentially large problems with a sequence of small changes, and makes it easier for a pattern language to evolve over time:

> Since the patterns are independent, then you can change one at a time, and they can always get better, because you can always improve each pattern, individually.
>
> (Alexander, 1979, p. 345)

Each pattern is thus a cluster of the forces in the domain of the systems that are being generated with the language. Developing a pattern language requires identifying key structures in the domain of implementation and articulating those structures as patterns that build on each other.

## *Example (architectural pattern language):*

Family of Entrances argues that entrances to a building complex or group of related buildings should be designed to be broadly similar and arranged so that each is visible from all others. If this is not done, then a person arriving at the building is likely to have trouble finding which entrance they need.

The forces addressed by Family of Entrances are as follows (Alexander *et al.*, 1977, pp. 499-502):

- Each building in a group of buildings or each section in a building complex needs an entrance.
- Each entrance needs to be distinguishable from others in the group.
- The collection of entrances needs to be recognizable as such.

These forces all relate to being able to easily identify a particular entrance within a collection of entrances. They are thus semantically related and relate to a common structure: the collection of entrances in a group of buildings or building complex.

Entrance Transition addresses the problem of making a person feel comfortable when they enter a building. The forces it addresses are as follows:

- People behave differently in public and private places.
- People need time and space in which to adjust their behavior from public to private and vice versa.
- People want their houses and, to some degree, other buildings, to be private domains.
- People expect the street to be a public domain.
- An entrance marks a transition from public to private space.
- Physical changes help create psychological changes.
- The experience of entering a building affects how you feel inside the building.

These forces all relate to facilitating a shift in behavior from public to private and vice versa. The common structure that they relate to is the particular entrance being built.

Although both Family of Entrances and Entrance Transition relate to creating entrances for buildings, none of the forces addressed by Entrance Transition pertain to distinguishing an individual entrance within a collection of entrances. Similarly, none of the forces addressed by Family of Entrances relate to smoothing the transition between the space inside and outside a building. The interaction of forces between the patterns is thus not significant, and these patterns thus cluster forces.

### *Example (C++ Idioms Pattern Language):*

Handle/Body splits a class into interface (handle) and implementation (body) classes, providing the convenience of scope-based memory management without tying the entity concerned to (program) scope.

The forces addressed by Handle/Body are as follows (Coplien, 2000a):

- C++ provides private and public sections as a mechanism to separate implementation from interface.
- A C++ class declaration makes the class implementation visible but inaccessible.
- Changing class implementation forces unnecessary recompilation of client code, even if the changes are made to private data.

These forces are semantically related because they all relate to separating interface and implementation in a C++ context. They are structurally related because they all pertain to a class structure that provides separation of interface and implementation.

Counted Body adds reference counting to a Handle/Body context. The forces addressed by Counted Body are as follows (Coplien, 2000a):

- C++ defines assignment as a deep copy; it uses recursive member-by-member assignment with copying as the termination of the recursion.
- Copying bodies costs significant resources.
- Pointers and references can be used to avoid copying, but they do not address the problem of garbage collection and "leave a user-visible distinction between built-in types and user-defined types".
- While sharing bodies might appear to be a good solution, "it is usually semantically incorrect if the shared body is modified through one of the handles".

These forces are semantically related because they all concern how to do reference counting in a Handle/Body context. They are structurally related because they relate to a class structure that builds on Handle/Body to do reference counting.

Although both Handle/Body and Counted Body are about memory management, and Counted Body assumes a Handle/Body set up as its context, none of the forces in Handle/Body relate to reference counting, and none of the forces in Counted Body relate to separation of interface and implementation. The interaction of forces between patterns is thus not significant, and is much lower than the interaction within the patterns. The patterns thus cluster related forces.

### *Example (CHECKS pattern language):*

Whole Value argues that specialized values that are meaningful in the application domain should be constructed and used for message arguments and as input/output units. For example, the duration of a contract should be specified in units of weeks or days, rather than as an integer.

The forces addressed by Whole Value are as follows:

- Literal values offered by the language and standard objects used as values will not correspond to meaningful quantities in a business domain.
- When quantifying a domain model, people want to use the most fundamental units possible.
- Communication between program and users needs common units.

These forces are semantically related because they all relate to the *fundamental units* used in the domain model. The forces also relate to a common structure: the units used to quantify the domain.

Exceptional Value addresses the problem of categorizing user input values that do not fit into the range of attributes covered by Whole Value. For example, if "agree" and "strongly agree" are two typical responses to a query, an answer such as "illegible" will not be able to be quantified in the domain model. Exceptional Value suggests that "one or more distinguished values [be used] to represent exceptional circumstances", rather than extending the range of attributes covered by Whole Value. Exceptional Value incorporates into the domain model a place for missing data that may appear at a later stage.

The forces addressed by Exceptional Value are as follows:

- Anticipating every possible business category in the domain hierarchy can be confusing and difficult.
- There will be a normal range of answers and other answers that are more exceptional.
- There needs to be a place in the class hierarchy for exceptional values.

These forces are semantically related because they all focus on categorizing the range of values in the domain. The forces all pertain to some classification structure for the values.

Although both Whole Value and Exceptional Value relate to quantities used by the domain model, none of the forces addressed by Exceptional Value pertain to identifying what units to use to quantify the domain. Likewise, none of the forces in Whole Value address how to categorize the range of values in the domain. The interaction of forces between the patterns is therefore not significant. The patterns thus cluster related forces.

### *Example (biological system):*

An animal confronted with a dangerous situation will often require rapid or unusually vigorous movement, heightened awareness, and extra strength. Rather than providing systems that are capable of sustained performance at this heightened level most animals have systems that can be temporarily switched into "emergency mode", with each system coordinated so as to provide an overall "super" being able to respond to the danger by either fighting or fleeing. The entire response is called the Fight or Flight response and is triggered by broadcasting a "danger" signal that every system responds to independently.

Key forces in Fight or Flight include the following:

- The response involves all systems in the animal acting simultaneously.
- The response need not last for long, but it must be rapid and immediate.
- Dangerous situations don't occur very often; most of the time, "normal" response mechanisms are adequate.

These forces all have to do with the immediacy of the response time. The common structure they relate to is the structure of interrelated systems in the animal's body that enable the broadcast mechanism that is the Fight or Flight response.

During periods of activity, animals consume nutrients and generate waste products. Rather than provide a system that is capable of sustained and continuous operation, most animals adopt an extended daily rest and recovery period during which required nutrient supplies are replenished and waste products removed. Conveniently, the recovery period is synchronized with the natural day/night cycle. This common arrangement might be called the Go to Sleep pattern. Forces in this situation include the following:

- Recovery takes time.
- Recovery requires nutrient supply.
- Recovery is more efficient if a system is not being actively stimulated.
- Most animals are optimized for performance either during the night or day but not both.
- The response involves all systems in the body.

These forces all relate to how to maintain the body in an operative state. The common structure they all connect to is the structure of interrelated systems that enable recovery to happen by sending the appropriate system an appropriate message at an appropriate time.

Although Fight or Flight and Go to Sleep both represent whole being responses to situations, none of the forces in Fight or Flight relate to recovery and none of the forces in Go to Sleep relate to responding immediately to an urgent situation. The interaction of forces within the patterns is much greater than that between patterns, and the patterns thus cluster related forces.

## *Related Patterns:*

*Resolution of Forces* shows how to create structure that holds a cluster of forces in tension.

## Local Symmetries

*If you need to understand how patterns are related*
*Then connect them according to the ways they break symmetry.*

### Context:

You have a collection of patterns, each of which solves a key problem in a domain. At least one of those patterns is not connected to other patterns in the language, or even if all the patterns are connected, you may feel that some connections are missing.

### Problem:

Patterns by themselves cannot solve complex problems. You need to connect them so that they form a language.

### Forces:

- System structure is defined by symmetry: "One of the great themes of the past century's mathematics is the existence of deep links between geometry and symmetry." (Stewart and Golubitsky, 1992, p. 4)
- Preserving symmetry preserves stability: "generally, the lowest energy state of an assembly is a symmetrical one" (Blundell and Srinivasan, 1996).
- Local symmetry, rather than global symmetry, is critical to a system's stability. A system needs more fine structure – more local symmetries – to be stable:

  > In general, a large symmetry of the simplified neoclassicist type rarely contributes to the life of a thing, because in any complex whole in the world, there are nearly always complex, asymmetrical forces at work – matters of location, and context, and function – which require that symmetry be broken.
  >
  > (Alexander, 2002a, p. 187)

  > … saying these patterns are alive is more or less the same as saying they are stable.
  >
  > (Alexander, 1979, p. 118)

- Each pattern exists in the context of other patterns and smaller-scale patterns complete larger-scale ones (Alexander *et al.*, 1977, p. xiii).
- While the cluster of forces in each pattern is independent, the patterns themselves work together and are interrelated:

> Each pattern encompasses, and contains, the forces which it has to deal with; and there are no other forces in the system. Under these circumstances, each event which happens is resolved. The forces come into play, and resolve themselves, within the patterns as they are.
>
> Each pattern helps to sustain other patterns.
>
> The quality without a name occurs, not when an isolated pattern occurs, but when an entire system of patterns, interdependent, at many levels, is all stable and alive.
>
> …
>
> So, somehow, this system of patterns, which I have loosely sketched, forms the basis for what is needed … and these patterns are a system; they are interdependent. It is true that each one can be explained, in its own terms, as an isolated thing, which is needed to resolve certain forces. But, also, these few patterns form a whole, they work as a system.
>
> (Alexander, 1979, pp. 130-131, 133)

### *Solution:*

Consider the scale of the patterns, which relates to the size of the artifact being shaped by the application of the pattern. Connect the patterns so that smaller-scale patterns break the symmetry of the larger-scale patterns to which they are connected.

To connect an isolated pattern, find a larger scale pattern that provides a context such that the pattern you are considering elaborates the symmetries provided by that context, thus preserving the overall structure of the larger pattern. Connect the isolated pattern below the larger pattern in the language. If there are several patterns that provide a suitable context for the isolated pattern then use domain knowledge to choose the right one or to decide whether more than one should be connected.

The new structure added to the language by the previously isolated pattern should create local symmetries; it should replicate and refine symmetry created by the solution structure of the pattern above. Then it will strengthen or refine the existing, overall structure created by the other patterns, making the overall structure more "whole".

Continue connecting patterns until all are connected. Since a pattern language is organized according to the levels of scale of the patterns (Fincher, 1999), it has a partial ordering, and the order in which you connect the patterns *per se* does not influence the resulting overall structure.

### *Resulting Context:*

The interrelationships between patterns in the language become clear, and the language develops an overall shape. Gaps in the language can now be identified, as can sequences of patterns potentially useful to developers. Significant domain knowledge can, however, be required to identify which symmetries are important and should form the basis for the connections in the language; a lack of appropriate domain knowledge will result in poor connections between patterns.

### *Rationale:*

Smaller-scale patterns elaborate larger-scale patterns by a process of differentiation (Alexander, 1979, pp. 369-370). Differentiation is not a process of combining preformed parts to make a whole, but rather one in which the whole gives birth to its parts by splitting, like the development of an embryo (Alexander, 1979, pp. 370-371). Differentiation of structure goes hand in hand with specialization: each pattern creates structure that is "specialized" to balance a given system of forces. Since patterns are transformations that break symmetry (Stewart and Golubitsky, 1992, p. xviii), smaller-scale patterns will elaborate the structure of larger-scale patterns by transforming existing, latent symmetries in the larger-scale patterns and expressing those symmetries more locally. The pattern language structure should reflect this relationship.

Czarnecki and Eisenecker (2000, pp. 1-16, 60-64, 131-142) affirm the importance of differentiation to system structure. They note that undifferentiated components are too large to be effective units of reuse, and that classes and objects alone are too small to have system impact. Alexander makes the same point in Building Complex (1977, pp. 468-472): "… the more monolithic a building is, and the less differentiated, the more it presents itself as an inhuman, mechanical factory." Alexander argues that it is important that building structures reflect the different social identities of people who use the buildings. In buildings that do so, a person's experience of using the building is more likely to be positive, and their interactions with other people in the building more likely to be regarded as personal, rather than impersonal.

### *Example (architectural pattern language):*

Family of Entrances addresses the problem of how to easily identify entrances in a collection of related buildings or a building complex. The solution it proposes is to make each entrance distinguishable by some small feature, but also visible from other entrances and broadly similar. A particular entrance can then be easily found (Alexander *et al.*, 1977, pp. 499-502).

Despite the need for multiple building entrances to look and feel similar, and to be placed in a way that creates local symmetries in the building structure, sometimes one entrance of a family functions as the main entrance to a building, while the others are alternate entrances. Main Entrance elaborates on Family of Entrances and suggests that in order to make the difference in function of the main entrance clear, it needs to stand out in some way from the others. One way in which this can be done is to add to the main entrance so that it juts out beyond the building line. The positional symmetry of the family of entrances is then preserved, and the main entrance is still visible from all other entrances identifiable as part of the family. The symmetry of look and feel, however, is broken. The main entrance no longer looks so broadly similar to other entrances. Main Entrance transforms Family of Entrances by breaking symmetry, because it has some of the symmetry of the other entrances – it still retains the positional symmetry – but it no longer has the look and feel symmetry. Main Entrance thus creates local symmetries.

Figure 5-10 on page 134 shows a small language that Alexander uses to create a garden that people enjoy using. The language incorporates, among other patterns, Half-Hidden Garden, Garden Growing Wild, Courtyards Which Live, and Tree Places. These four patterns are near the top of Alexander's language, and the connections between them are illustrated as part of the language structure diagram. Half-Hidden Garden is the largest-scale pattern in the language. Garden Growing Wild and Courtyards Which Live both build on Half-Hidden Garden, and Tree Places builds on both Garden Growing Wild and Courtyards Which Live.

Half-Hidden Garden says that people won't use a garden that is too public or too isolated. You should therefore make a garden "half-hidden" - partly isolated, partly open.

Garden Growing Wild elaborates on Half-Hidden Garden by giving some details about the garden. It suggests that the garden should not be precisely planned but allowed to grow more naturally, to meander. Most planned gardens tend to be globally symmetric. Unplanned nature, with the wiles and ways of its vines and branches, has much more broken symmetry (local symmetry) than a planned garden. Garden Growing Wild elaborates on Half-Hidden Garden by transforming an empty garden into one with structures that create local symmetries.

Courtyards Which Live also elaborates on Half-Hidden Garden. It says that this particular Half-Hidden Garden will be a courtyard, which is usually an enclosed area,

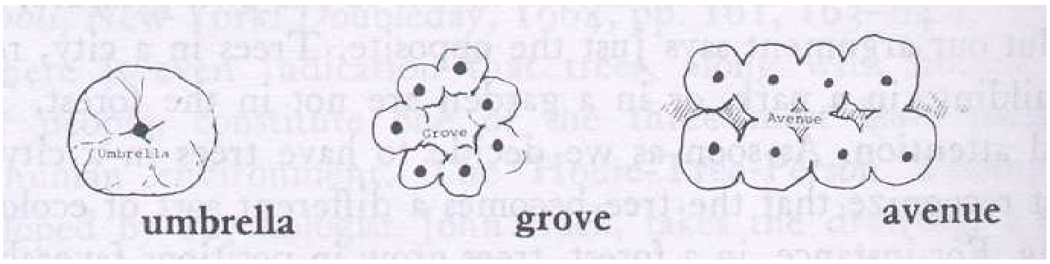

**Figure 5-3: Examples of structures that create Tree Places (Alexander *et al.*, 1977, p. 800).**

but with a view to the outside, and with paths running across it from several entrances. The elaboration again creates local symmetries. Partially enclosing the area of the garden adds a local symmetry, defined entrances to the garden add local symmetries, and paths across the space of the garden, running between entrances, add further local symmetries.

Tree Places elaborates on both Garden Growing Wild and Courtyards Which Live, because the tree places are formed in both those spaces. Tree Places points out that trees must not be added to a garden "without regard for the special places they create". Trees should be added so that they form "enclosures, avenues, squares, groves, and single spreading trees toward the middle of open spaces. And shape the nearby buildings in response to trees, so that the trees themselves, and the trees and buildings together, form places which people can use."

Tree Places elaborates on Garden Growing Wild and Courtyard Which Lives by creating local symmetries. It suggests adding trees to the space created by those patterns so that the trees form smaller, definable spaces within the garden. Figure 5-3 shows how Alexander illustrates some of the kinds of spaces he is suggesting. These illustrations clearly show the symmetries created by the suggested spaces.

The structure of the language, as shown in Figure 5-10 on page 134 reflects the local symmetries created by the patterns. Garden Growing Wild and Courtyards Which Live are both connected to and below Half-Hidden Garden, and both elaborate on Half-Hidden Garden by creating local symmetries. Tree Places is connected to and below both Garden Growing Wild and Courtyards Which Live, and it elaborates on the structure created by both those patterns by creating local symmetries.

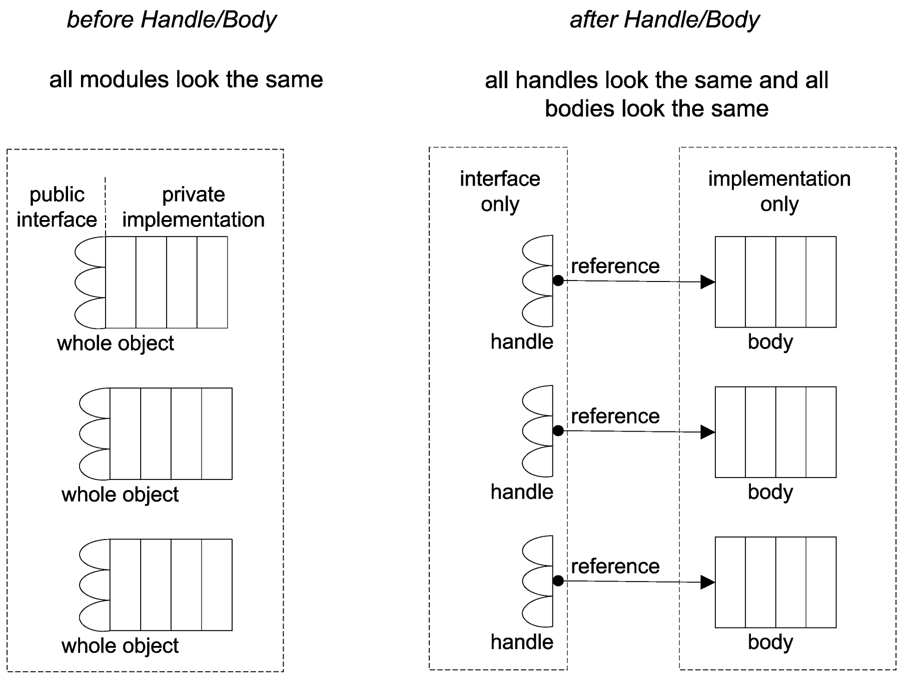

**Figure 5-4: Handle/Body transforms a design-time class into two distinct implementation classes, Handle and Body. In the process, the type symmetry of the original class is redistributed, with the Handle expressing the interface symmetry and the Body the implementation symmetry.**

### *Example (C++ Idioms Pattern Language):*

Handle/Body splits a design-time class into two implementation classes, one of which is the interface (Handle) and the other the actual implementation (Body).

Handle/Body creates local symmetry by redistributing type symmetry, as shown in Figure 5-4. The original type (before Handle/Body is applied) expresses symmetry over a related set of objects (the objects of that type) in the system. That is, there are elements of the interface and elements of the implementation that are invariant across all the objects of the type. Specifically, the interface member functions, their parameter and return types, their pre- and post-conditions, are all the same. Similarly, the implementation structure of all of the objects is the same.

Handle/Body redistributes this symmetry in two parts: the handle, and the body. One part (the Handle) carries the original interface symmetry, and the other (the Body) carries the original implementation symmetry. Each of these parts expresses its own local symmetry – all handles of the type look the same, and all bodies of the type look the same. Conceptually, the Handle and Body are each still part of one whole object: the Handle/Body object. Each of these local symmetries derives from the original symmetry; the original symmetry is preserved, but at a more local level.

Using Handle/Body creates a situation where more than one Handle can potentially refer to the same Body. For example, the top two handles in Figure 5-5 both refer to identical bodies, which represents a source of inefficiency. Counted Body addresses this problem by adding a reference counter to the body objects, thus allowing more efficient memory use while still preserving the independence of the Handle/Body objects.

In the idioms language, Counted Body is connected to Handle/Body, indicating that the language structure of the idioms language reflects the symmetry breaking relationships between Handle/Body and Counted Body. Counted Body builds on Handle/Body by preserving the existing type symmetry but elaborating on it more locally. The symmetry that Counted Body preserves has to do with the relationship between Handle/Body and client objects. Each client of a Handle/Body object "sees" the object in the same way, because all of the object functionality is accessed through

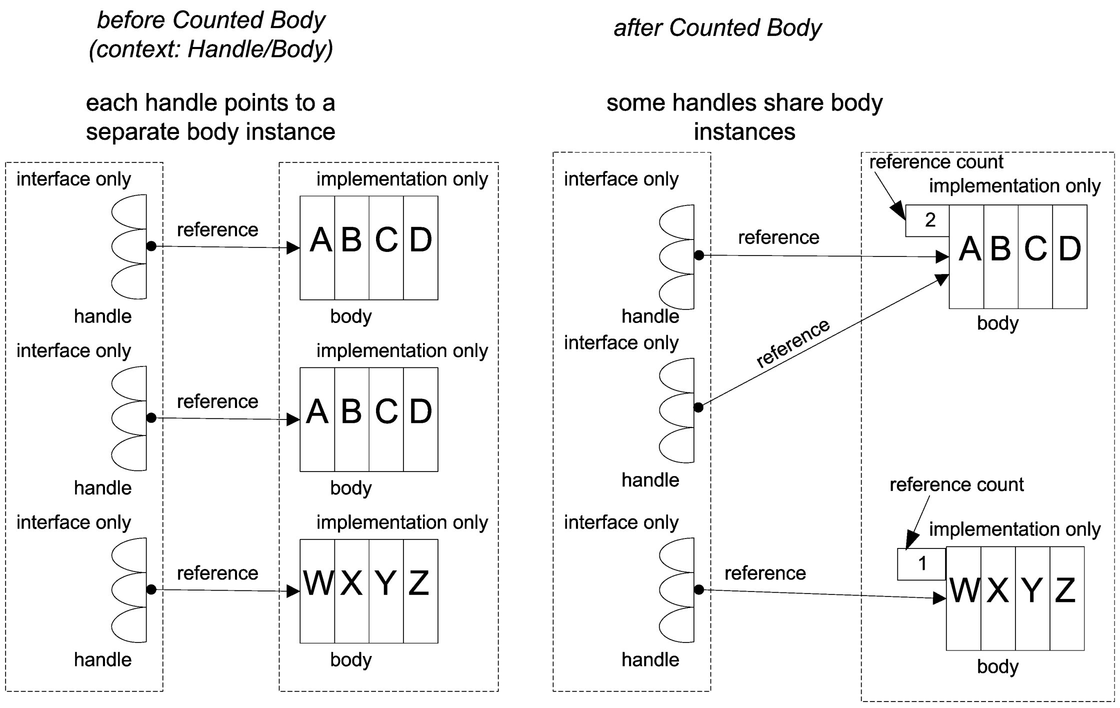

**Figure 5-5: Counted Body transforms Handle/Body by maintaining a reference count and allowing multiple handles to refer to the same body. This preserves the symmetry whereby each client of a Handle/Body object views it (locally) in the same way, regardless of which handle provides access.**

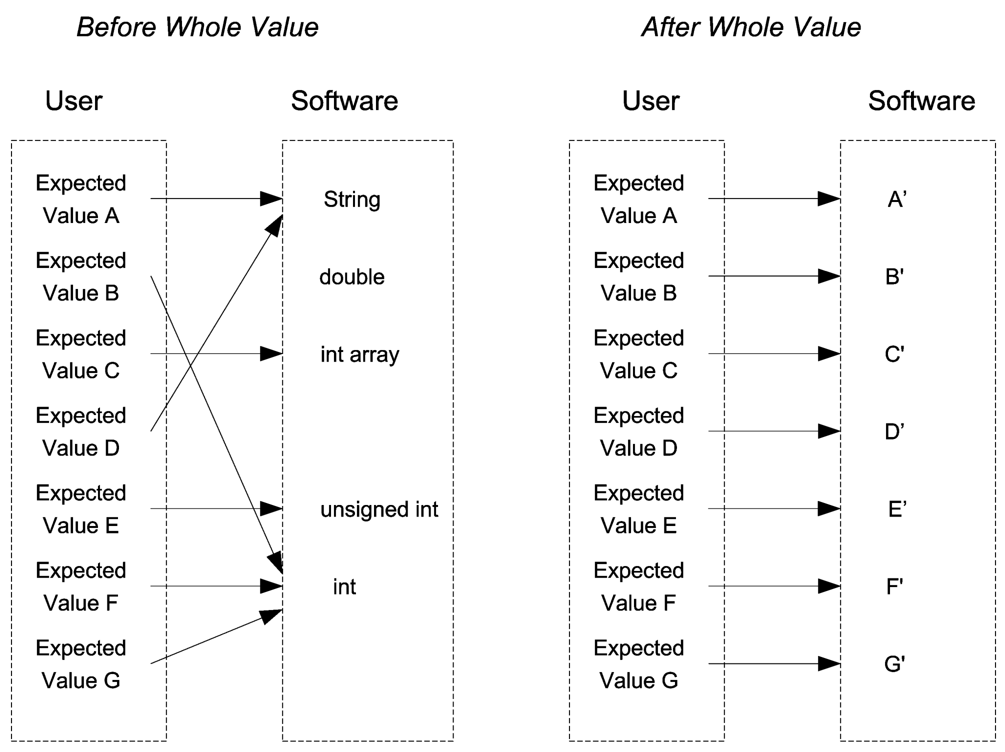

**Figure 5-6: Whole Value brings about a correspondence between structures in the user/application domain and structures in the software domain.**

the one Handle. From the client's (local) point of view, all objects look the same. But from the system (global) point of view, only the handles that share a body are the same; a handle for one body is different from a handle for a different body. Thus Counted Body preserves the symmetry created by Handle/Body (from the client's point of view) while enabling greater efficiency (from the system's point of view).

### *Example (CHECKS pattern language):*

Whole Value (Cunningham, 1995) points out that while programmers often seek to quantify values in a domain model using the most fundamental units possible, this is neither necessary nor useful. It is much more useful to quantify domain values using specialized values that have meaning in the domain. This facilitates smooth and proper communication between different parts of a program, and between a program and its users.

User interface software enables communication between two different domains: that of a business or application area, and that of a particular software development paradigm. The fundamental structures of those domains may well be different. Whole

Value points out that a user interface will best facilitate communication between the software and application domains if it stores values from the application domain in software structures that replicate structures in the application domain, as shown in Figure 5-6. This is a symmetry; the pattern encompasses two different domains, and brings them together by replicating the structures of one domain in the other.

Exceptional Value relates to categorizing values that fall outside the range defined by a pattern like Whole Value. The need for a pattern like Exceptional Value arises because including all business possibilities in a class hierarchy is often impractical and confusing, and yet there needs to be a place in the software domain structure for all possible values from the application domain. Exceptional Value suggests using "one or more exceptional values to represent exceptional circumstances", and processing exceptional values in exactly the same way as expected values are processed, as shown in Figure 5-7.

Exceptional Value creates local symmetries by reinforcing the symmetries of a pattern like Whole Value. When Whole Value is used, structures are created in the software domain that mirror the key structures in the application domain. Each

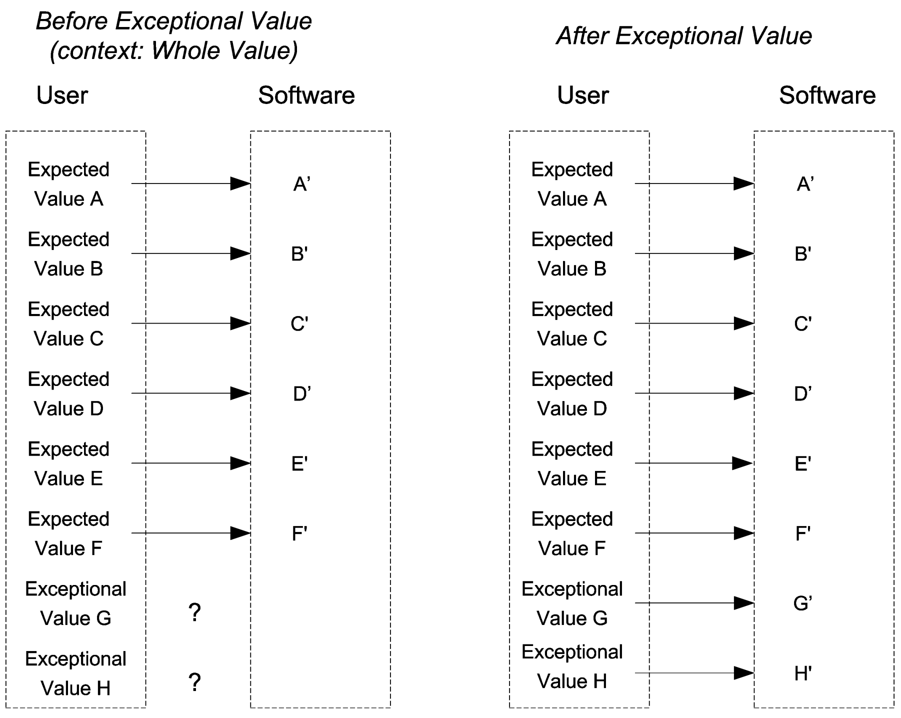

**Figure 5-7: Exceptional Value deals with exceptional or unexpected values in the same way that expected values are dealt with.**

transformation of a value from application to software domains is thus a local symmetry; the structure remains the same across different domains. Exceptional Value adds additional local symmetries of this kind, but for exceptional values, or values that do not fit within standard categories, rather than for the standard categories defined by Whole Value.

Echo Back (Cunningham, 1995) also addresses an issue related to communication between a program and its users, pointing out that data from the domain entered by the user should be stored in the domain model and then echoed back to the user. This keeps the user informed of whether the domain model has accepted or rejected input, and how the domain model has interpreted input, without breaking the input entry cycle. If communication between the user and the domain model only happens in one direction, from the user to the domain model, the effectiveness of communication is decreased. The user may not become aware of input errors until a large batch of data has been entered and that batch is reviewed, if at all.

Echo Back is slightly smaller in scale than Whole Value and connected to it, because instead of focusing on the overall structure of the whole domain, Echo Back assumes an overall structure created by a pattern such as Whole Value, and addresses a problem with one aspect of that structure. There is no language diagram for the CHECKS language to illustrate the connection. Instead, the connection is indicated by an assumption explicitly stated in Echo Back that it adds to a Whole Value context: "Field and cell widgets will be able to construct and deliver Whole Values to the domain model." So Echo Back builds on Whole Value. Further, the same symmetry expressed in Whole Value, whereby domain structure is mirrored in the program structure, is replicated in Echo Back, only this time the replicated structure is a communication from domain to program that is replicated in the other direction. The connection between Whole Value and Echo Back in the CHECKS language can thus be explained in terms of local symmetries.

### *Example (Half-Object Plus Protocol pattern):*

Half-Object Plus Protocol (HOPP) represents a transformation of a system in which a concept that has been implemented in exactly one address space splits across two or more address spaces. The object implementing that concept is transformed such that it comprises two (or more) "half-objects" plus a protocol that manages interaction between the half-objects. The protocol and half-objects together make up the HOPP object, as shown in Figure 5-8.

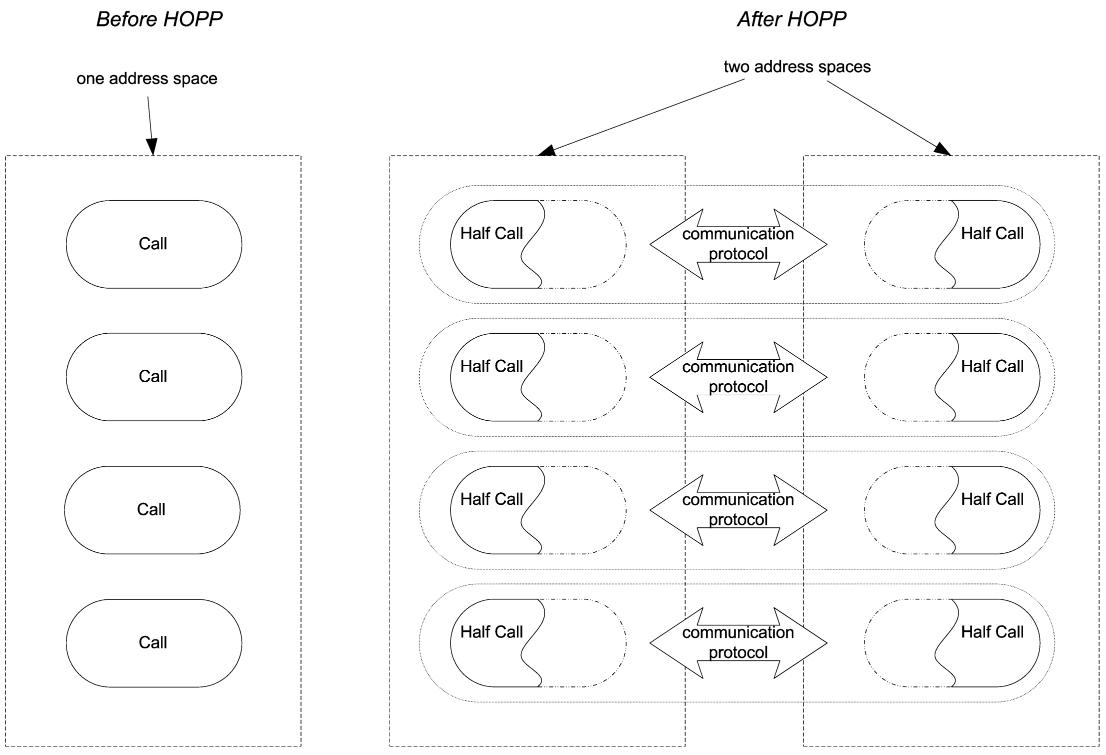

**Figure 5-8: An example of HOPP for Call objects. The key to recognizing the symmetry breaking in HOPP is to recognize that each half-object is created as like the original full object as possible.**

The HOPP pattern builds on Whole Value (Cunningham, 1995), because it quantifies domain values using specialized values that have meaning in the domain. A specialized Phone Call class, for example, might be constructed to allow Phone Call objects to represent phone calls in a telecommunications system. In a system implementing Whole Value, the relationship between a particular object and each of its client objects is the same: each of the client objects "sees" the given object in the same way, because all of the object's functionality is in the same place. This sameness of relationship between object and *all* client objects is a symmetry; the invariant is the relationship between object and client, and what changes is the structure of the client itself. This symmetry might be lost when a concept is split across object spaces, but HOPP preserves it, by transforming the object into a HOPP object and redistributing symmetry *inside* what is now a HOPP object to create local symmetries. Each half-object is created as like the original as possible, even if that means duplicating some functionality. Clients can interact with each half-object as if it were the original, whole object.

When forces require a concept to be implemented across multiple address spaces, HOPP thus indicates how to change the system to allow the distribution, yet still preserve the symmetry of the original system as much as possible. HOPP provides an implementation that preserves a conceptual symmetry of relationship between object and client, by creating local symmetries in the system implementation (Coplien, 1998a; b). Within the context of a pattern language, HOPP would thus be connected to a pattern like Whole Value, because it breaks the symmetry of Whole Value.

### *Example (biological system):*

In many species, the embryo is initially spherically shaped. As it develops, the embryo changes shape, but not randomly; the embryo of a given species follows a predictable pattern of development. Many major events in embryo development break symmetry, and this symmetry breaking is a key to the predictability of the developmental process (Harris, 2004).

Cleavage is the name given to the initial period of embryo development and is an example of a pattern in embryo development that breaks symmetry. During cleavage, an embryo repeatedly doubles the number of cells of which it is made up, as shown in the top row of Figure 5-9. Each cell of the embryo repeatedly splits into two copies of itself; this process is called mitosis. Throughout cleavage, there is no growth of the embryo. The embryo splits into ever smaller cells (Kimball, 2004), creating local symmetries; the structures of the original zygote are replicated on a smaller and smaller scale, and still form part of an identifiable whole.

Gastrulation, shown in the second row of Figure 5-9, is another example of a pattern in embryo development that breaks symmetry. During gastrulation, cells in the embryo reposition themselves to form the three primary germ layers, each of which has a special role in building the complete animal (Kimball, 2004). During gastrulation the embryo develops a definite anterior and posterior, and left and right orientation. Gastrulation replaces axial symmetry with bilateral symmetry; the embryo moves from having an infinite number of planes of reflection symmetry to having only one plane of reflection symmetry (Harris, 2004). The embryo thus retains some, but not all of its original symmetry, and the full reflectional symmetry is found more locally in smaller areas within the embryo.

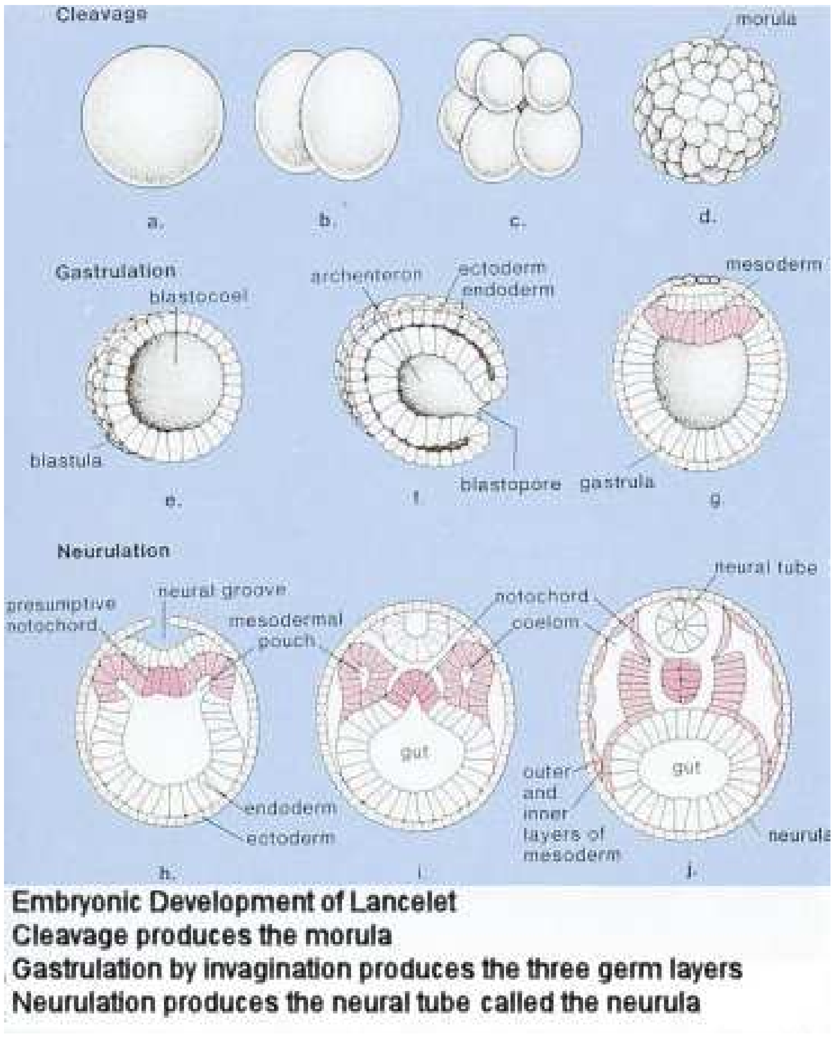

**Figure 5-9: Three stages in embryonic development: cleavage, gastrulation, and neurulation.**

## *Related Patterns:*

*Levels of Scale* shows what to do when the difference in scale between patterns is too great for a connection to be made. *Cross Linkages* highlights the importance of the overall language structure being more than a simple, tree-like structure.

## Resolution of Forces

*If you need to balance a cluster of forces*
*Then build a structure that holds the forces in tension.*

### *Context:*

You have identified the pattern that a new force belongs with, but that force cannot be resolved within the pattern as it stands.

### *Problem:*

You need to resolve the new cluster of forces.

### *Forces:*

- The forces within a cluster need to hold each other in tension to produce a stable structure. Alexander notes that each pattern is stable, or self-sustaining, to the degree that it is capable of "containing its own forces, and keeping them in balance" (Alexander, 1979, p. 126).
- For forces to be held in tension, they must pull in different directions.
- Each cluster of forces relates to a single, common physical form: "The ultimate object of design is form." (Alexander, 1964, p. 15)
- You will not know that you have resolved forces until you have built a physical form that resolves them.
- Balancing forces will cause new forces to arise in the patterns.

### *Solution:*

Balance the cluster of forces by building a structure that holds the forces in tension. This structure will become a pattern. Use domain knowledge to determine when forces are balanced. If you cannot balance the clustering using domain insight, then look for a missing force. If a missing force cannot be found, then the clustering is probably wrong; discard the cluster of forces and try again.

### *Resulting Context:*

The new pattern includes the new force in the set of forces it resolves. However, if the identified forces are incomplete or inaccurate, or your domain knowledge is insufficient to identify how best to balance the forces, the pattern may be weak.

### *Rationale:*

In a very pragmatic sense, you need to write a solution for your patterns, and this is where you do it.

You will not be able to work out simply by theoretical means whether or not your clusters of forces are the right ones, and whether they will resolve into good patterns. Some clusters of forces cannot be balanced using cursory domain information, and there is no simple rule with which to verify that a pattern does resolve its forces. Alexander (1979, p. 286) notes that it requires an intangible mix of analytical and practical understanding, and that "we must rely on feelings more than intellect". You will have to create the patterns – create the structures – before you can work out whether they resolve forces:

> We cannot decide whether a misfit has occurred either by looking at the form alone, or by looking at the context alone. Misfit is a condition of the ensemble as a whole, which comes from the unsatisfactory interaction of the form and context.
>
> (Alexander, 1964, p. 96)

Creating the patterns may also reshape your understanding of the forces:

> A well-designed house not only fits its context well but also illuminates the problem of just what the context is, and thereby clarifies the life which it accommodates. Thus Le Corbusier's invention of new house forms in the 1920's really represented part of the modern attempt to understand the twentieth century's new way of life.
>
> …
>
> At the time of its invention the geodesic dome could not be calculated on the basis of the structural calculations then in use. Its invention not only solved a specific problem, but drew attention to a different way of thinking about load-bearing structures.
>
> In all these cases, the invention is based on a hunch which actually makes it easier to understand the problem. Like such a hunch, a constructive diagram will often precede the precise knowledge which could prescribe its shape on rational grounds.
>
> It is therefore quite reasonable to think of the realization as a way of probing the context's nature, beyond the program but parallel to it. This is borne out, perhaps, by the recent tendency among designers to think of their designs as hypotheses. Each constructive diagram is a tentative assumption about the nature of the context. Like a hypothesis, it relates an unclear set of forces to one another conceptually; like a hypothesis, it us usually improved by clarity and economy of notation. Like a hypothesis, it cannot be obtained by deductive methods, but only by abstraction and invention. Like a hypothesis, it is rejected when a discrepancy turns up and shows that it fails to account for some new force in the context.

(Alexander, 1964, pp. 91-92)

### *Example (architectural pattern language):*

Family of Entrances addresses the problem of easy identification of building entrances within a collection of related buildings or building complex. The solution it proposes is twofold. It suggests making each entrance distinguishable by some feature, such as a number, or being on a corner, or having a particular kind of tree or bush near it. It also suggests laying out the collection of entrances so that they "form a family". Laying out the entrances so that they form a family requires the following:

> 1. They [the entrances] form a group, are visible together, and each is visible from all the others.
>
> 2. They are all broadly similar, for instance all porches, or all gates in a wall, or all marked by a similar kind of doorway.
>
> (Alexander *et al.*, 1977, p. 502)

This solution balances the forces addressed by Family of Entrances because it identifies a way of making an entrance both distinguishable from yet similar to other entrances at the same time.

Entrance Transition addresses the problem of how to make a person feel comfortable when they enter a building. The solution it proposes is as follows:

> Make a transition space between the street and the front door. Bring the path which connects street and entrance through this transition space, and mark it with a change of light, a change of sound, a change of direction, a change of surface, a change of level, perhaps by gateways which make a change of enclosure, and above all with a change of view.
>
> (Alexander *et al.*, 1977, p. 552)

This solution balances the forces addressed by Entrance Transition. Those forces include that people behave differently in public and private places, and that they want their homes and to some degree other buildings to be relatively private, while expecting the street to be a public place. Recognizing that the experience of entering a building affects how a person feels inside that building, Entrance Transition creates a physical structure that provides time and space in which psychological transitions can happen, and helps trigger those transitions with changes in the physical space, such as a change of light or sound.

### *Example (C++ Idioms Pattern Language):*

The solution proposed by the Handle/Body pattern is as follows:

> [split] a design class into two implementation classes. One takes on the role of an identifier and presents the class interface to the user. We call this first class the *handle*. The other class embodies the implementation and is called the *body*. The handle forwards member function invocations to the body.
>
> (Coplien, 2000a)

This solution balances the forces clustered by Handle/Body. The first of those forces is that C++ provides public and private class sections as a mechanism for facilitating separation of interface and implementation. Yet, as the second and third forces note, this mechanism can be ineffective, because changes to class implementation still require recompilation of client code and class implementation is still visible in a C++ class declaration. Handle/Body uses the existing class construct to provide a way of separating interface and implementation that is not provided for in the language syntax. However, it both allows implementation changes to be made independent of the client and hides implementation code from the client, balancing the forces.

Counted Body offers the following solution to the problem of doing reference counting for Handle/Body objects:

> Add a reference count to the body class to facilitate memory management; hence the name "Counted Body."
>
> Memory management is added to the handle class, particularly to its implementation of initialization, assignment, copying, and destruction.
>
> It is incumbent on any operation that modifies the state of the body to break the sharing of the body by making its own copy. It must decrement the reference count of the original body.
>
> (Coplien, 2000a)

This solution balances the forces clustered by Counted Body. The first two of those forces note that assignment in C++ is defined in terms of bit copying and is thus expensive, so programmers tend to want to redefine assignment more efficiently themselves. The third force points out that instead of using assignment, programmers can use pointers and references to implement more efficient copying, but that the use of pointers and references has a significant cost, creating a "user-visible distinction between built-in types and user-defined types" and leaving the problem of garbage collection still to be dealt with. The fourth force highlights a problem with one way of attempting to implement assignment more efficiently for Handle/Body objects; modifying a shared body through one of its handles is semantically incorrect. Counted Body balances these forces by providing a memory management approach that both takes care of garbage collection and can work with user-defined assignment operations, provided that Counted Body operations that modify body state make their own copy of the body object and adjust the reference count appropriately. The independence of forces in Handle/Body and Counted Body also means that Counted Body can be added to a language without affecting Handle/Body.

### *Example (CHECKS pattern language):*

Whole Value addresses the problem of what kind of units to use in a programming language to model a domain, and offers the following solution:

> [construct] specialized values to quantify [the] domain model and use these values as the arguments of their messages and as the units of input and output.
>
> (Cunningham, 1995)

This solution balances the forces clustered by Whole Value. The first of those forces notes that values in the programming language lack meaning in the domain. Yet, as the second force notes, people tend to use those values anyway; people tend to want to use the most fundamental units possible, regardless of domain meaning. The third force notes that effective communication between program and users requires common units. Whole Value balances these forces by providing values in a programming language that represent meaningful quantities in a domain. This both enables effective communication between program and users and allows meaningful domain quantities to be represented, while helping the programmer resist the temptation to use the most fundamental units possible to quantify the domain model.

Exceptional Value addresses the problem of categorizing the range of input values to a program, providing the following solution:

> Use one or more distinguished values to represent exceptional circumstances. Exceptional values should either accept all messages, answering most of them with another exceptional value, or reject all messages (with `doesNotUnderstand`), with the possible exception of identifying protocols like `isNil` or `printOn:`.
>
> (Cunningham, 1995)

This solution balances the forces clustered by Exceptional Value. The first of those forces is the difficulty of anticipating every possible business category. Yet, despite this difficulty, there still needs to be a way of storing values that don't fit in an identified and expected category; this is the third force. The second force points out that values can at least be categorized as normal and exceptional. By categorizing values into normal and exceptional, Exceptional Value provides a way of catering for unexpected values without having to anticipate every unexpected value, balancing the forces. The independence of forces across Whole Value and Exceptional Value means that even though Exceptional Value builds on Whole Value, it can be added to the language without affecting Whole Value.

### *Example (biological system):*

An animal in a situation of immediate danger responds by immediately broadcasting an alarm for individual systems in the body to respond to as appropriate. This pattern is known as Fight or Flight. In more detail, the hormone adrenalin is released into the bloodstream and this stimulates the liver to convert glycogen (stored energy) to glucose (usable energy), the eyes to dilate the pupils so that vision is sharpened, the heart to increase the pulse rate so that oxygen flow to the blood is increased, and so on. The Fight or Flight response differs from the usual response to stimulus in that it uses a broadcast mechanism, rather individual messages from the nervous system. Fight or Flight addresses forces that relate to the immediacy of response needed, and yet the need for the response to be across all systems in the body. The solution to Fight or Flight allows for a number of systems to respond simultaneously and immediately to a crisis situation, thus balancing the forces.

An animal that needs to rest responds by gradually reducing the functioning of systems in the body to a minimal level, enabling recovery. This pattern is called Go to Sleep. In more detail, the lack of input to muscles decreases their tension, the lack of activity causes the metabolic rate to fall, the lack of need for oxygen causes the heart rate to fall, the lower metabolic rate causes the temperature to fall, and brain rhythms change. This response balances the forces clustered by Go to Sleep by making time for recovery during a time when an animal is not geared for optimal performance.

### *Related Patterns:*

*Differentiation* shows how to create new structure by refining existing structure, *Aggregation* shows how to incorporate new structure by molding it to fit the existing structure, even if it is different, and *Common Ground* shows how to incorporate new structure by drawing out underlying similarities in existing structure.

## Levels of Scale

*If the difference in scale between patterns is too great*
*Then add patterns that make the difference smaller.*

### Context:

You have a collection of interconnected patterns of different levels of scale that form a language.

### Problem:

You cannot connect patterns with the right differences in scale between them; how to go from one pattern to another is unclear.

### Forces:

- Patterns in a language must be of different levels of scale (Fincher, 1999).
- The complex systems that pattern languages generate have patterns on many different levels of scale, and the language needs to reflect this: "Multiscale descriptions are needed to understand complex systems." (Bar-Yam, 2000-2005)
- Patterns at different levels of scale are interrelated: "Each pattern then, depends both on the smaller patterns it contains, and on the larger patterns within which it is contained." (Alexander, 1979, p. 312)
- If connected patterns are too different in scale then the language will lack coherence:

> ... *I must also make sure that the patterns below a given pattern are its principal components.*
>
> There must not be too many patterns underneath a given pattern. Consider HALF-HIDDEN GARDEN this time. There is one corner of the garden which will be the SUNNY PLACE; another place will be an OUTDOOR ROOM perhaps; there is a need for trees, forming a place; there is the character of the garden as a whole; GARDEN GROWING WILD; there is the relation between the garden and the street, in detail; PRIVATE TERRACE ON THE STREET. There is the possibility of the relation between the house and the garden: covered perhaps by GREENHOUSE .... There is the character of flowers in the garden: perhaps RAISED FLOWERS, and the need for vegetables and fruit: VEGETABLE GARDEN and FRUIT TREES . . . .
>
> But not all of these need to be directly underneath HALF-HIDDEN GARDEN. The reason is that some of them embellish *each other*. For example, GARDEN GROWING WILD, which gives the gardens global character, is itself filled out and completed by RAISED FLOWERS and VEGETABLE GARDEN; and TREE PLACES is itself filled out by FRUIT TREES. These patterns which can be "reached" through another pattern, do not need to appear directly below the pattern HALF-HIDDEN GARDEN.

> *It is essential to distinguish those patterns which are the principal components of any given pattern, from those which lie still further down.*
>
> If I have to make a HALF-HIDDEN GARDEN and I can understand it as a thing which has three or four parts, 1 can visualize it, and begin to create one, for myself, in my own garden.
>
> But if the HALF-HIDDEN GARDEN has twenty or thirty patterns, all equally its parts, I will not be able to imagine it coherently.
>
> (Alexander, 1979, pp. 320-321)

- Symmetries in a design must gradually unfold to give it order:

> the presence of this "objective beauty" was closely tied to the presence of symmetries and subsymmetries that balance the use of contrasting space, light, and color to form fields of visual centers. Feelings of beauty and order would increase when these visual centers unfolded recursively at multiple hierarchical levels of granular scale throughout a design (much like fractals).
>
> (Appleton, 2000)

### *Solution:*

Make sure the connections between patterns reflect differences in scale of roughly 2:1 to 7:1. The scale of a pattern relates to the size of the artifact being shaped by the application of the pattern. If necessary, look for patterns whose scale is appropriate to fill the gaps that are too large in scale in the language. Those new patterns will break the symmetry of larger-scale patterns, and will have their symmetry broken by smaller-scale patterns.

### *Resulting Context:*

The way that smaller patterns relate to larger patterns becomes clear; system developers will be given better guidance about how smaller patterns build on larger patterns. The language as a whole provides a good picture of the kinds of artifacts that can be generated with the language. If, however, you do not use your domain knowledge carefully, you may create artificial or contrived patterns simply because they happen to fill a perceived gap in the language.

### *Rationale:*

Alexander (2002a, pp. 145-150) points out that artifacts that exhibit certain jumps in levels of scale have much more of the essential, structural quality that he calls life than artifacts that don't. Specifically, if jumps in scale between patterns are deliberate and evenly spaced and of a magnitude from 10:1 to 2:1, then the patterns of an artifact intensify each other and increase the life of the whole artifact. Salingaros goes further and argues that three architectural laws can be extracted from physics and mathematics by looking at how nature is put together:

1. Order on the smallest scale is established by paired contrasting elements, existing in a balanced visual tension.
2. Large-scale order occurs when every element relates to every other element at a distance in a way that reduces the entropy.
3. The small scale is connected to the large scale through a linked hierarchy of intermediate scales with scaling factor approximately equal to $e = 2.718$.

(Salingaros, 1995)

The entropy of a design relates to the innate human ability to perceive connections. Salingaros argues that when all components of a design interact harmoniously the design will resonate with people's innate perception of connections and this corresponds to states of least entropy. Salingaros further notes that pattern languages can be represented as multilevel hierarchical systems, and that in such systems

> Even though disconnected lower-level patterns can work without necessarily forming a higher-level pattern, such a system is not cohesive, because it exists on only one level. Each level in a complex hierarchical system is supported by the properties of the next-lower level.
>
> (Salingaros, 2000)

Alexander also points out that the measured jumps in scale he argues are key to good design are intrinsic to biological organisms. Limburg, for example, notes that in ecological systems, connections tend to be strongest across similar scales:

> "In general … there will be tighter coupling among processes and components with similar rates and overlapping spatial scales."
>
> (Limburg *et al.*, 2002)

Patterns of the same scale work together but this relationship is implicit in the language structure and the way the language is supposed to be used (patterns of the same size applied consecutively), rather than being shown in the drawn connections between patterns.

### *Example (architectural pattern language):*

In Alexander's language for creating a garden, shown in Figure 5-10 on page 134, if the patterns in-between Half-Hidden Garden and Garden Seat were not part of the language, it would be hard to envision what kind of garden could be created. It could be argued that Garden Seat breaks the symmetry of Half-Hidden Garden, creating local symmetries simply because it adds structure to what is otherwise an almost blank state - a garden that will be half-hidden but has little other specified structure. But if you follow the language through from Half-Hidden Garden to Garden Growing Wild, then to Tree Places, then to Fruit Trees, and finally to Garden Seat, you have a much better picture in your mind of what kind of garden is being envisaged; the language is much more coherent.

### *Example (C++ Idioms Pattern Language):*

Handle/Body splits a class into interface (handle) and implementation (body) classes, providing the convenience of scope-based memory management without tying the entity concerned to (program) scope. Homogeneous Addition shows how to begin to build an efficient type system for addition and suggests that the `add` method of a subclass should only deal with operands of its own subclass. If the idioms language contained Handle/Body and Homogeneous Addition, but not any of the patterns in-between – Handle/Body Hierarchy and Algebraic Hierarchy – it would be difficult to see how Handle/Body and Homogeneous Addition were related. It could still be argued that Homogeneous Addition broke the symmetry of Handle/Body, in the sense that each method represents a local symmetry, but it would be difficult to see why homogeneous addition required a Handle/Body context; why the pattern could not build on many other patterns. Handle/Body Hierarchy and Algebraic Hierarchy provide necessary mid-scale structures that fill out the relationship between Handle/Body and Homogeneous Addition. Handle/Body Hierarchy shows how to create a hierarchy of Handle/Body objects, and Algebraic Hierarchy shows how to place basic algebraic operations in an inheritance hierarchy, bearing in mind that those operations depend equally on two operands.

### *Example (CHECKS pattern language):*

Whole Value argues that values corresponding to key values in the application domain should be created in the software domain. Visible Implication argues that derived or redundant quantities implied by entered values should be computed and displayed to the user. Given just these descriptions, it is hard to see exactly how these two patterns could be connected. It can be argued that Visible Implication breaks the symmetry of Whole Value by making communication happen from system to user, as well as from user to system, but when Echo Back is added to the language the symmetry breaking connections become much clearer. Echo Back displays back to the user any information (any whole value) written into the domain model. It breaks the symmetry of Whole Value, creating local symmetries by replicating the communication from user to domain in the opposite direction every time it occurs. Visible Implication then breaks the symmetry created by Echo Back and Whole Value. Visible Implication computes values defined as key by Whole Value and implied by entered information, and echoes them back to the user, so that Echo Back now applies to computed values as well as entered values.

### *Example (biological system):*

Alexander gives examples of the presence of a continuous range of structures at different scales is widespread throughout many natural systems:

> The tree: trunk, limbs, branches, twigs. The cell: cell wall, organelles, nucleus, chromosomes. A river: bends in the river, tributaries, eddies, pools at the edge. A mountain range: highest mountains, individual peaks, surrounding foothills, still smaller sub-hills. Limestone: large particles, smaller particles, smallest particles each in the interstices of larger ones. The sun and solar system: planets and their orbits, satellites and their orbits. A molecule: component complexes, individual atoms and ions, neutrons, protons, electrons. A flower: individual flower heads, center and petals, sepals, stamens, pistils.
>
> (Alexander, 2002a, p. 246)

### *Related Patterns:*

New patterns may be fleshed out with *Differentiation*, *Common Ground*, or *Aggregation*. The connections to the new patterns will fit the overall scheme of *Local Symmetries*. If the language becomes too crowded it can be cleaned out with *The Void*.

## Cross Linkages

*If the pattern language structure cannot express essential complexity*[2]
*Then look for other linkages that express that complexity.*

### *Context:*

You are developing the structural interconnections between patterns in a pattern language that generates complex systems.

[2] Essential complexity refers to a situation where all reasonable solutions to a problem must be complicated (and possibly confusing) because the "simple" solutions would not adequately solve the problem. (Wikimedia Foundation Incorporated (2006) *Essential Complexity: from Wikipedia, the Free Encyclopedia*, <http://en.wikipedia.org/wiki/Essential_complexity>, accessed November 10, 2007.

### *Problem:*

You need to make the language rich enough to be able to generate complex systems.

### *Forces:*

- A simple hierarchical structure will not solve complex problems (Alexander, 1988).
- System structure needs to be in alignment with fundamental domain structure (Winograd and Flores, 1986, p.53).
- Containment or specialization is not the only kind of structural relationship in many domains.
- Every pattern is connected to other patterns: "Each pattern sits at the center of a network of connections which connect it to certain other patterns that help to complete it." (Alexander, 1979, p. 313)

### *Solution:*

Connect patterns expecting that some patterns will break the symmetries of more than one larger-scale pattern. For a pattern to break the symmetries of more than one pattern means that it replicates and refines symmetry created by the combined solution structures of more than one pattern.

### *Resulting Context:*

The language has the richness necessary to generate systems that solve complex problems rather than just trivial ones. However, adding unnecessary interconnections will make the language excessively complex, which could make it less effective.

### *Rationale:*

The structure of complex systems cannot be articulated in a simple, tree-like hierarchy, because elements of such systems sometimes overlap, rather than being either wholly contained in or disjoint from other system elements. Overlap is, in fact, an essential generator of complex structure. A better representation of a complex system is a semi-lattice, which allows its elements to overlap. The main constraint on a semi-lattice is that "when two overlapping sets belong to the collection [semi-lattice], the set of elements common to both also belongs to the collection" (Alexander, 1979).

Alexander (1988) argues that the structures of a city are too complex to be captured in a simple, tree-like structure. Further, he argues that cities whose design is based on such a tree structure are unsuccessful and will fail in their ultimate aim to improve the human condition. In contrast, successful cities have a more complex structure of interactions or cross-linkages that can be represented by a semi-lattice.

Many of Alexander's patterns reflect the need for systems generated with those patterns to exhibit cross linkages. One such pattern is Scattered Work (1977), in which Alexander argues that work zones need to be decentralized and woven in with residential zones, because a complete separation does not reflect the integrated nature of people's lives. People need interconnections between the many different aspects of their lives.

Alexander notes that the structure of a pattern language must be complex enough to generate complex systems and describes the connections between patterns as reflecting more than a simple hierarchy, with patterns building on potentially more than one other pattern:

> Suppose we use a dot to stand for each pattern, and use an arrow to stand for each connection between to patterns. Then [A• →• B] means that the pattern A needs the pattern B as part of it, in order for A to be complete; and that the pattern B needs to be part of the pattern A, in order for B to be complete.
>
> If we make a picture of *all* the patterns which are connected to the pattern A, we see then that A sits at the center of a whole network of patterns, some above it, some below it.
>
> [diagram omitted]
>
> Each pattern sits at the center of a similar network.
>
> *And it is the network of these connections between patterns which creates the language.*
>
> (Alexander, 1979, p.313)

Senge (1990, pp. 114-135) reinforces the importance of identifying connections between patterns in his work on systems thinking. He argues that understanding how the parts of a complex system interact is critical because it is out of these interactions that the behavior of the system as a whole arises. Simply identifying parts of a complex system is not sufficient to understanding the system.

The structure of many software systems is more complex than can be expressed in a simple hierarchy. Consider, for example, the common programming problem of decoupling an abstraction from its implementation so that the two can vary independently. One application where this problem occurs is in the development of a hierarchy for real and complex numbers. Real is a specialization of Complex, so class Real inherits from class Complex. But having Real as a subtype of Complex means

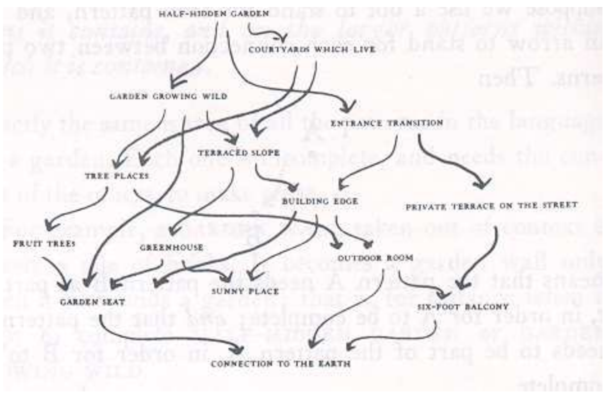

**Figure 5-10: A diagram of the structure of a language for creating a garden. Note that several patterns have more than one arrow coming to them from other (larger-scale) patterns, indicating the complexity of the relationships between patterns.**

that each Real number has an unnecessary imaginary part, because the inheritance transformation comprises both interface and implementation. One way of solving this problem is to separate the interface and implementation inheritance so that a number's implementation is not tied to its interface. The existing inheritance hierarchy is split into two hierarchies, one of which represents interface inheritance and the other implementation inheritance. The hierarchies are connected by each object from the interface hierarchy maintaining a pointer to an object from the implementation hierarchy, whose particular type can be determined at run-time.

The above situation highlights a case where a simple hierarchy does not provide the complexity of structure needed to solve a problem. The solution creates a cross link: the one hierarchy is split into two, and the pointer acts as a cross link between them. The above problem and solution are outlined in the Bridge pattern (Gamma *et al.*, 1995).

## *Example (architectural pattern language):*

As shown in Figure 5-10, in the diagram of a language for a Half-Hidden Garden, some patterns build on more than one other pattern. For example, Tree Places builds on both Garden Growing Wild and Courtyards Which Live because the latter two patterns together define the space of the garden. Trees can be added to the garden to create Tree Places that are partly contained in the Garden Growing Wild and partly contained in the Courtyards Which Live areas of the garden.

### *Example (C++ Idioms Pattern Language):*

The structural diagram for the idioms language, shown in Figure 5-11, illustrates the links (Coplien, 2000a) between patterns in the language. The fact that some patterns build on more than one other pattern (that is, have arrows from more than one other pattern to themselves indicates that the structure of the language cannot be represented as a simple hierarchy.

For example, Promote and Add implements heterogeneous addition where the operands are of different subtype within the same hierarchy. It builds on both Promotion Ladder and Homogeneous Addition, using Promotion Ladder repetitively to promote the object of more specific type to the more general type and Homogeneous Addition to then perform the addition. The language structure diagram shows arrows to Promote and Add from both Promotion Ladder and Homogeneous Addition.

### *Example (CHECKS pattern language):*

While no language structure diagram is provided for the CHECKS pattern language

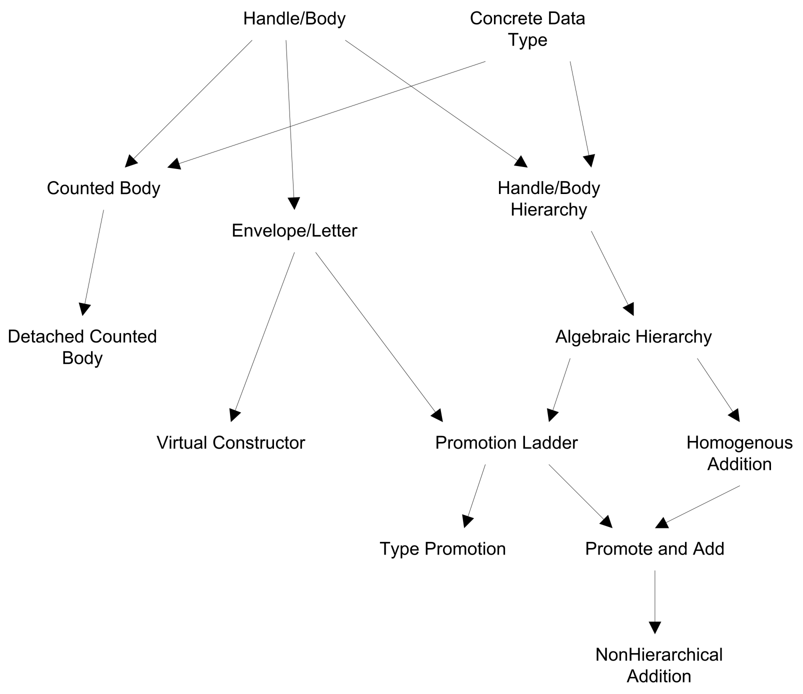

**Figure 5-11: a diagram of the language structure of the idioms language, based on (Coplien, 2000a). Patterns such as Counted Body and Handle/Body Hierarchy build on more than one pattern.**

(Cunningham, 1995), the descriptions of the patterns indicate that interconnections between them are more complex than a simple hierarchy. For example, Meaningless Behavior builds on both Whole Value and Exceptional Value. Whole Value suggests working with entities in the domain model that have meaning in the domain. Exceptional Value shows how to create space in the domain model for unexpected input values that have no meaning in the domain. Meaningless Behavior suggests not attempting to attach meaning to errors when that meaning is actually unpredictable from the information at hand. Instead, Meaningful Behavior suggests expecting input/output widgets to recover from any failure and continue processing. Meaningless Behavior is most powerful in a context where both Whole Value and Exceptional Value are used, because these latter two patterns constrain the situations that Meaningless Value needs to deal with.

### *Example (biological system):*

Storch and Gaston (2002) note that while "general macroecological patterns are quite simple, the relationships between them are complex and not straightforward". They analyze ecological complexity on different scales of space and time, identifying key macro-ecological patterns and analyzing their interconnections. Storch and Gaston note that some patterns must reflect a multidimensional relationship that is not simply hierarchical. For example, one pattern they identify is Species - Area Relationship, which notes that species richness tends to increase with area. They note that this pattern must be connected in some way to all patterns relating to commonness and rarity. Patterns related to commonness and rarity include Spatial Variation in Abundance, which notes that "abundance of a species is highly unequally distributed within its range", and Local-Regional Species Richness Relationship which notes that local species richness tends to correlate with regional species richness.

### *Related Patterns:*

*Local Symmetries* and *Levels of Scale* guide the connections between patterns.

## Differentiation

*If incorporating a new force requires finer-scale structure*
*Then create a new pattern that breaks the symmetry of an existing pattern.*

### *Context:*

You have identified the pattern most closely related to a new force that needs to be incorporated in the language, but the new force cannot be successfully incorporated within that pattern.

### *Problem:*

In order to incorporate the new force you need finer-scale structure.

### *Forces:*

- Maintaining the stability of the existing structure is important, and system structure is defined by symmetry (Stewart and Golubitsky, 1992, p. 4).
- The largest-scale patterns in a language are of greatest morphological importance with respect to the artifact being built. Smaller-scale patterns therefore need to conform to the structure of larger-scale patterns (Alexander, 1979, p. 384).
- Design can be viewed as "a sequence of acts of complexification" (Alexander, 1979, p. 370). New structure is always laid down on the basis of existing structure.
- The structure of patterns in the language must have reached a certain level of detail in order to be useful, because the language must have enough levels of scale to be compatible with the application domain.
- Particular patterns must become specialized for particular tasks (Alexander *et al.*, 1977, p. xii).

### *Solution:*

Add a new pattern whose structure elaborates that of the identified pattern by breaking symmetry. The new pattern should express latent symmetry in the original pattern more locally.

### *Resulting Context:*

Your language is able to express enough detail about structure to be useful in generating artifacts, and you will have been able to incorporate that detail in a way that fits in with, rather than compromises, the overall structure of the language. Sometimes, however, simply introducing finer-scale structure is not enough, and a re-clustering of forces is actually what is needed.

### *Rationale:*

Alexander informally defines differentiation as smaller patterns building on larger patterns by a process of elaboration, rather than one of combining pre-formed parts. It is like the process by which an embryo develops: a whole gives birth to its parts by splitting. It is a process in which existing symmetry is preserved as much as possible. As system structure is defined by symmetry, preserving symmetry preserves structure, and thus promotes stability of system structure.

Montgomery noted the importance of preserving symmetry in the context of development of telecommunications software:

> Consider a distributed system in which the HOPP pattern is present. If the system needed to expand from, say, two to three sites, it could do so easily by transforming the existing HOPP structure from two half-calls to three, and so on for further adaptation. The relationship between client and existing "half-objects" does not need to change, and the new client object relationship is the same as that for existing client/object pairs. It is this preservation of symmetry that enables this kind of incremental growth, by allowing the growth to be based on existing system structure.
>
> Making the model symmetric, if you can, leads to great simplifications. [A system I worked on] started out symmetric, no notion of originating and terminating terminal process, just terminal processes. The call protocols were all symmetric. This made it very easy to take the next step of going to multi-party calls …
>
> (Montgomery, 1998)

In terms of the language, adding the new pattern differentiates the local structure produced by the language, because it introduces a new pattern that replicates and refines an existing pattern.

### *Example (architectural pattern language):*

Mosaic of Subcultures argues that the homogeneous nature of many modern cities is ultimately unhealthy, and that cities should be comprised of "a vast mosaic of small and different subcultures, each with its own spatial territory, and each with the power to create its own distinct life style" (Alexander *et al.*, 1977, p. 50). However, even if a city starts of as number of separate, distinct communities, as the city grows those different subculture areas inevitably impinge on each other and are likely to lose their distinctive character unless that character is sustained in some way. At the point where distinctive subcultures begin to impinge on each other, the need to nurture distinctive cultures becomes a new force that needs to be addressed. One way of addressing this force is to create physical boundaries between subcultures; this pattern is Subculture Boundary. Subculture Boundary highlights the need for boundaries, suggests how big such boundaries should be, and gives pointers for creating them. Subculture Boundary is smaller in scale than Mosaic of Subcultures, dealing only with boundary areas in a broader architecture defined by the overall mosaic idea. The addition of Subculture Boundary to Alexander's pattern language at the point when maintaining separate, different subcultures becomes an issue is an example of *Differentiation*.

### Example (C++ Idioms Pattern Language):

Handle/Body shows how to split a class into interface (handle) and implementation (body) classes in a way that provides effective memory management. The forces it addresses relate to the separation of interface and inheritance in a C++ context. One issue that arises out of using Handle/Body relates to memory usage; it is potentially wasteful not to allow multiple handles to reference the same body. At the point at which memory usage becomes critical, this force needs to be incorporated into the language. It is a force that clearly belongs in a Handle/Body context, but the Handle/Body pattern does not resolve it. The force can be resolved by creating a new, more specialized pattern – Counted Body – that works within a Handle/Body context to do reference counting. It solves the problem of providing multiple handles for the same body by adding a reference count to the body class and memory management features to the handle class. The addition of Counted Body to the idioms language at the point when reference counting becomes a critical issue is an example of *Differentiation*.

### Example (CHECKS pattern language):

Whole Value argues that values used for message arguments and as input/output units should have meaning in the application domain. If necessary, specialized values should be constructed in the solution domain. Inevitably, though, some user input values will not fit into the range covered by Whole Value. For example, an answer such as "illegible" cannot quantified in a context where answers such as "definitely agree" and "depends on context" are typical. At the point where it becomes necessary to represent unusual, non-typical values, the need for their inclusion represents a new force that needs to be incorporated into the language. While the force does relate to representing application domain values in the solution domain, incorporating it into the existing solution structure for Whole Value would require being able to predict all possible unusual values and extending the range of attributes covered by Whole Value ad infinitum to cater for them. Such a solution is impractical; Whole Value thus does not resolve the new force. The force can be resolved by creating a new, more specialized pattern – Exceptional Value – that works within a Whole Value context to categorize user input values that do not fit into the range of attributes covered by Whole Value. Rather than extend the range of attributes covered by Whole Value, Exceptional Value suggests creating one or more distinguished values catering for all exceptional circumstances and incorporating these exceptional values into the Whole Value structure. The addition of Exceptional Value to the CHECKS language at the point when the inclusion of unusual values becomes important is an example of *Differentiation*.

### *Example (biological system):*

The process of cell division in which chromosomes are equally partitioned and replicated into two identical groups is called mitosis. As most biological organisms develop, cells initially simply divide and replicate. At this stage, all cells are called "stem cells" because they are undifferentiated; all cells appear alike and each cell has the potential to become any more specialized kind of cell. If the only biological patterns in existence were patterns to do with cell division, however, many biological structures would not exist. At some point during the development process, there is a need for cells to specialize for particular tasks in order for the organism's structure and functionality to develop to an appropriate level. At the point at which this need becomes operative, this need represents a new force that can be addressed by a Stem Cell Specialization pattern. Stem cells begin to progressively specialize, with cells initially being able to be distinguished organ, skin-and-gut, or nerve cells. Each of those groups of cells then further specializes: for example, skin-and-gut cells specialize into cells for the skin, or cells for the lining of the gut. The transformation whereby stem cells specialize represents a refinement of existing structure rather than a fundamental modification of existing structure:

> The cells of the body differ in structure and function not because they contain different genes, but because they express different portions of a common genome.

(Campbell, 1996, p.985)

If a biological pattern language had been developed without Stem Cell Specialization, then the addition of Stem Cell Specialization at the point when the need for cells to specialize for particular tasks became an issue would be an example of *Differentiation*.

### *Related Patterns:*

*Differentiation* is closely related to *Aggregation*. *Local Symmetries* tells how to connect patterns in a language. ***The Void*** shows how to clean out patterns if further specialization is ineffective.

## Aggregation

*If a different kind of structure needs to be added to the language*
*Then mold it so that it specializes existing larger-scale structure.*

### *Context:*

You have identified the pattern most closely related to a new force that needs to be incorporated in the language, but the new force cannot be successfully incorporated within that pattern.

### *Problem:*

In order to accommodate the new force you need a solution structure that represents an alternative to that provided by the identified pattern.

### *Forces:*

- A system will need to evolve over time so that new structure will need to be added. Systems grow; pattern languages, being a system, also grow. But they must grow in an orderly way if they are to remain stable.
- The need for finer structure does not drive all system growth. Sometimes new structure is needed for other reasons, such as to cope with environmental change or increased demands. While such new structure ought to be molded to fit with existing structure, and at some level of scale will represent a specialization of existing structure, in the context in which the change is made new structure may

not always be a specialization of existing structure, but rather a new kind of structure.

- Change does not happen in isolation; change to one part of a system will affect other parts.
- Yet, you want to localize the effects of change as much as possible.

### *Solution:*

Use the pattern that most closely related to the new force as the basis for developing a new pattern. The two patterns will represent alternative solutions to a particular problem, and will build on the same pattern(s). When creating the new pattern, discard forces that contradict the new force that you need to incorporate, and modify the solution structure appropriately.

### *Resulting Context:*

The language has a new pattern that enables new solution strategies and fits within the broader context of the language. Because the new pattern is a refinement of a larger level of scale, the stability of the language is maintained. Sometimes, however, the new force cannot be incorporated into a pattern that is a refinement of an existing pattern; in that case, a reclustering of forces is needed.

### *Rationale:*

A pattern language is a continually evolving artifact. As it is being developed, new patterns will be added:

> We must … invent new patterns, whenever necessary, to fill out each pattern which is not complete.
>
> (Alexander, 1979, p.323)

Not every pattern will be easily identifiable as refining some slightly larger-scale pattern. Some patterns will incorporate new structure into a language:

> We see now that there is a second, complementary process which produces the same results, but works piecemeal, instead.
>
> When a place grows, and things are added to it, gradually, being shaped as they get added, to help form larger patterns, the place also remains whole at every stage - but in this case the geometric volume of the whole keeps changing, because there is an actual concrete aggregation of matter taking place.
>
> (Alexander, 1979, pp. 491-492)

The key to effectively incorporating new structure is to remember that every act of growth is an act of repair in a broader context (Alexander, 1979, p. 480). So, for example, adding extensions to a house might add new structure to the house, but it is an act of repair to the larger environment that the house is part of:

> … suppose that you have built a small laboratory building.
>
> It has a kitchen, a library, four labs, and a main entrance. You want to add a fifth laboratory to it, because you need more space.
>
> Don't look for the best place right away. First, look at the existing building, and see what is wrong with it. There is a path where tin cans collect; a tree which is a beautiful tree, but somehow no one uses it; one of the four labs is always empty, there is nothing obviously wrong with it, but somehow no one goes there; the main entrance has no places to sit comfortably; the earth around one corner of the building is being eroded.
>
> Now, look at all these things which are wrong, and build the fifth lab in such a way that it takes care of all these problems, and also does, for itself, what it has to do.
>
> (Alexander, 1979, p. 481)

### *Example (architectural pattern language):*

Alexander's architectural language includes several patterns relating to how to build houses for different groups of people: House for a Small Family, House for a Couple, and House for One Person. A similar pattern not included in his language might be called House for a Student Community. Many adult students share houses together and have needs distinct from any of the other identified communities. While the pattern House for a Student Community addresses similar kinds of forces to the above patterns, it does not refine any of them; adding it to the pattern language represents the addition of new structure to the language. It can be added to the language by copying one of the other patterns, since it is similar enough to have some overlap, and modifying it appropriately. Then it could be added to the language by connecting it to a larger-scale pattern, probably the same pattern the others are connected to, and it will break the symmetry of that larger-scale pattern. Incorporating House for a Student Community into Alexander's pattern language is thus an example of *Aggregation*.

### *Example (C++ Idioms Pattern Language)*

At some point in the development of the idioms language, the author realized that some systems generated with the language needed to be able adapt Handle/Body code to implement reference counting, but without the access to Body code assumed by Counted Body. The new force addressed - that the body code is immutable - cannot be incorporated into the solution structure of Counted Body as it contradicts a force implicit in the solution to Counted Body - the force that the Body code can change. This suggests using *Aggregation* to create a new pattern using Counted Body as a starting point. The force that the body code can be modified is deleted from the new pattern and the force that the body code be immutable added. According to this change, Detached Counted Body ought to sit under Handle/Body in the idioms language. While this is not where the author put Detached Counted Body, he agrees that this is a reasonable adjustment (Coplien, 2003). Incorporating Detached Counted Body into the idioms language is thus an example of the kind of *Local Repair* where a new force cannot be balanced within an existing cluster of balanced forces. It is an example of *Aggregation*.

### *Example (CHECKS pattern language):*

Hypothetical Publication suggests that users ought to be able to release any number of hypothetical publications into the system in a controlled manner. Part of the reason for this is to make it more likely that the presented data is more thoroughly examined prior to publication. In the case where forecasting tools were not available, though, or were not well used, the importance of analyzing how data could be used would be hard to resolve within Hypothetical Publication. The lack of forecasting tools represents a new force and can be resolved with a pattern such as Instant Projection, which argues that it is worthwhile to attempt to anticipate how a publication will be used and project that information for consideration prior to publication. Instant Projection is not a specialization of Hypothetical Publication, but rather the two patterns complement each other and either or both can be used according to context. The two patterns both build on the larger-scale patterns Visible Implication, which suggests computing derived or redundant quantities implied by entered values, and Deferred Validation, which suggests delaying domain model validation until certain key points. Incorporating Instant Projection into the CHECKS language in a situation where forecasting tools become unavailable is thus an example of *Aggregation*.

### *Example (biological system):*

Social insects such as ants live in colonies where a mature colony has one or multiple queens and a collection of workers (Oster and Wilson, 1978). When a colony is first started, though, it may consist only of a queen who does all the work herself. It is only as more and more young are reared that the colony develops a structure in which particular ants become specialized for particular tasks. As particular ants become specialized for particular tasks, the structures of a colony change. Instead of Everyone Does Everything being the most efficient way to operate, Everyone Specializes becomes more efficient. Everyone Specializes is not a specialization of Everyone Does Everything; it is a different way of working. Incorporating this pattern into a language describing the structure of an ant colony thus represents *Aggregation.*

### Related Patterns:

*Aggregation* is closely related to *Differentiation*. *Local Symmetries* tells how to connect patterns in the language. *The Void* shows how to clean out patterns if further specialization is ineffective.

## Common Ground

*If several patterns have underlying similarities*
*Then reformulate the patterns into a more general, slightly larger-scale pattern, able to be applied in a greater variety of situations.*

### Context:

You have identified several patterns that closely relate to a new force that needs to be incorporated in the language.

### Problem:

The new force does not clearly relate most closely to one particular pattern.

### Forces:

- Some patterns address similar clusters of forces.
- Some patterns have a similar solution structure.
- Different patterns "… have underlying similarities, which suggest that they can be reformulated to make them more general, and usable in a greater variety of cases." (Alexander, 1979, p. 330)

### Solution:

Create a new pattern that draws out underlying similarity in the identified pattern and other similar, existing patterns. In particular, find two or more patterns of a similar level of scale that have underlying similarities because they address similar forces and have similar solution structures. The scale of a pattern relates to the size of the artifact being shaped by the application of the pattern. The new pattern should explicitly cluster the underlying forces that the other patterns have in common and it should build a structure that balances those forces.

### *Resulting Context:*

Patterns in the language are more focused; key forces across different contexts have been drawn into a single pattern. Recognizing underlying commonality between patterns also offers the potential to provide better jumps between scale, allow a language to be useful in a wider variety of languages, and even highlight connections between different languages. There are, however, may cases where patterns share forces but where they should not build on a common pattern. If your domain knowledge is not sufficient to recognize when the need for a common pattern exists, you may end up with "artificial" patterns that do not represent solutions to real problems.

### *Rationale:*

As Alexander (1979, p. 330) notes, some patterns identified in different contexts reflect the same structural configuration, yet each pattern is in a slightly different context with slightly different forces and different finer structure. Drawing out the underlying structural commonality in the patterns into a new pattern may help identify new contexts where that structure may be used. The new pattern may also add a necessary level of scale to an existing language, and identifying the new pattern may lead to additional, more specific, smaller-scale patterns also being identified.

### *Example (architectural pattern language):*

In their work developing the University of Oregon, Alexander and colleagues (Alexander, 1975, p. 131) noted that a department building that is just a collection of offices does not facilitate the development of community or the exchange of ideas. Department Hearth addresses this issue. It notes the need, within a departmental building, for a common area, close to well-used paths, where people feel comfortable to sit and have a coffee and chat. The pattern Family Room Circulation (Alexander, 1979, pp. 329-330) addresses a similar issue, but in the context of Peruvian houses. It recognizes the need for a room that allows those living in the house to interact easily

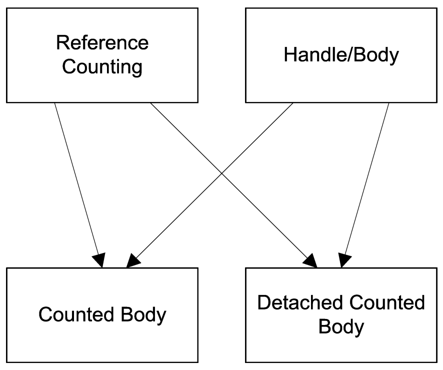

**Figure 5-12: A diagram showing the language structure of part of the Idioms Language were a Reference Counting pattern to be included in the language.**

with visitors without disrupting the privacy of the rest of the house. Tangent Paths again addresses a similar issue, but this time in the context of developing clinics. It notes the need for paths to pass tangent to social areas in order for those areas to be used.

Alexander notes that underlying each of these patterns is the same bundle of relationships; they each require "a common area at the heart of a social group, placed in such a way that people's natural paths passed tangent to this common area, every time that they moved in and out of the place" (Alexander, 1979, p. 330). Out of these patterns Alexander and colleagues identified the more general and slightly larger in scale pattern Common Area at the Heart.

### *Example (C++ Idioms Pattern Language):*

Both Counted Body and Detached Counted Body are patterns of similar level of scale. They could both be positioned in the idioms language directly underneath Handle/Body, since they can be thought of as representing alternative ways of building on Handle/Body. Both Counted Body and Detached Counted Body each cluster forces that primarily have to do some aspect of reference counting; the forces listed for Counted Body discuss how to best manage multiple handles for one body, and those for Detached Counted Body discuss how to do the same without modifying body code. Although the link to reference counting is not picked up in the idioms language as it stands, a Reference Counting pattern could be added to the idioms language, as shown in Figure 5-12. Together with Handle/Body, such a pattern would provide a helpful basis from which to implement Counted Body or Detached Counted Body. Adding the new pattern would be an example of *Common Ground.*

### Example (Checks Pattern Language):

Both Echo Back and Visible Implication assume a context in which the quantities used for data provided to a program have meaning in the application domain, and the program provides feedback to users in terms of those quantities. If Whole Value had not been included in the CHECKS language, one way of recognizing its importance is to realize that existing patterns such as Echo Back and Visible Implication assume a context in which the program needs to be able to operate using values meaningful in the application domain.

### Example (biological system):

Stewart and Golubitsky (1992) analyze various animal gaits. Among the quadruped gait patterns that they identify are Pronk, Trot, Pace, and Bound. In Pronk, all four legs move at the same time; it is like a four-legged hop. For a Trot, the left front and right back legs move together, and the left back and right front legs move together; diagonal leg pairs thus move together and in phase. Each diagonal pair of legs is out of phase with the other by half a phase period. In Pace, the left legs move together and in phase, as do the right legs, and each pair is half a period out of phase with the other. In Bound, it is the front legs that move together and in phase, as do the back legs, and each pair is half a period out of phase with the other.

As well as analyzing leg gaits, Stewart and Golubitsky seek to identify underlying commonality among all the gait patterns. They look for simple patterns capable of generating all the different leg gaits that they analyze and identify two key patterns characterizing such gaits, In-Phase and Out-of-Phase. Each of these patterns compares the motion of two legs, one leg from each of the leg pairs identified in the gaits above. In the case of In-Phase, both legs can be regarded as oscillators with the same waveform that can be interchanged without affecting system dynamics. In contrast, for the Out-of-Phase pattern, both legs can be regarded as oscillators with the same waveform but with a half-period phase difference.

### Related Patterns:

*Local Symmetries* and *Levels of Scale* show to connect patterns in the language. *The Void* shows how to clean out patterns if further specialization is ineffective.

## The Void

*If the overall language structure gets too tangled*
*Then clean it out to create space for more development.*

### Context:

The existing language structure has become convoluted and tangled.

### Problem:

How to further grow the language.

### Forces:

- Adding more local structure will make the language more convoluted.
- Structure that is all detail diffuses itself, destroying its own structure; "calm is needed to alleviate the buzz" (Alexander, 2002a, p. 225).

### Solution:

Excise the tangle by breaking some connections in the language, and possibly removing some patterns, thus creating space for new growth.

### Resulting Context:

The language will remain relevant over time as patterns and their interconnections change. New patterns will not be stifled by archaic ones; new patterns will have space to resolve and recluster forces as necessary. Choosing when to use The Void, however, requires significant domain knowledge. If you use it at the wrong time, or too frequently, you could destroy needed patterns and decrease system stability.

### Rationale:

Alexander (2002a, pp. 222-225) points out the importance of emptiness to good design. He argues that some spaces must be empty in order for the overall structure to be effective. For example, when using building materials, it is much more common for large amounts of one substance surrounded by small amounts of another to be economical and efficient, rather than equal amounts of two substances. He notes that a concept of emptiness, or a void, is central to many of the architectural patterns he describes, including Bathing Room, (1977, pp. 681-686)which provides a necessary calm space amidst the busy life of a household. Positive Outdoor Space (1977, pp. 517-523) also relates to the need for space. It describes the kind of shapes that buildings should create around them, by their shape, and argues that buildings that create convex spaces around them create a positive space - a void. Conversely, buildings that do not create convex spaces do not create any distinctly identifiable spaces around them. When a building creates positive outdoor space, that space works as a void, or space between buildings that people identify as distinct and feel comfortable in.

Natural processes also work by periodically "cleaning out" when things get too tangled. For example, forest fires regenerate forests, and plants and animals die to give space for new plants and animals to live.

A designed system should therefore be cleaned out when it becomes too tangled.

### *Example (architectural pattern language):*

Alexander points out that the pattern language he wrote with colleagues is continually evolving. As part of that process, some patterns are removed from the language:

> A group of us began to construct such a language eight or nine years ago. To do it, we discovered and wrote down many hundreds of patterns. Then we discarded most of these patterns during the years …
>
> (Alexander 1979, p.331)

For example, when designing new buildings for the University of Oregon, Alexander (1975) discarded many patterns from his original pattern language (1977), as well as incorporating some new patterns specific to the university context. Discarding the unnecessary patterns allowed designers to focus on those few patterns most important to the design at hand.

### *Example (C++ Idioms Pattern Language):*

In an algebraic hierarchy implemented in C++, an operator like addition might be implemented by providing each class with an addition method for all possible type combinations. A language for such an implementation might include pattern such as Simple Addition and Combining Different Types showing how to do addition in this way. But if the number of types catered for increased, then such a solution could quickly become unworkable, due to the combinatorial explosion in type combinations. A solution such as Homogeneous Addition would be more appropriate, but in order for the language to cleanly articulate such a solution, and especially if the new solution is deemed a better solution even for small numbers of types, the existing patterns relating to the addition operation would need to be removed to give space for the new patterns to develop into.

### *Example (CHECKS pattern language):*

Many existing patterns potentially build on patterns in the CHECKS language. Null Object (Walker, 2003), for example, is a special case of Exceptional Value. An Exceptional Value "will either absorb all messages or produce Meaningless Behavior … A Null Object is one such Exceptional Value."

Many other patterns are related to Null Object in various ways. Since every Null Object has no internal state that could change and multiple instances would act exactly the same, Singleton (Gamma *et al.*, 1995, pp. 127-136) can be used to implement Null Object. Similarly, Flyweight (Gamma *et al.*, 1993, pp. 195-206) can be used for more efficient implementation where multiple null objects are implemented as instances of a single Null Object class. As part of the Strategy (Gamma *et al.*, 1993pp. 315-324) pattern, Null Object can be used to represent the strategy of doing nothing. The State (Gamma *et al.*, 1993, pp. 305-313) pattern often includes representation of a state in which the client should do nothing; Null Object is often used to represent this state. Iterators (Gamma *et al.*, 1993, pp. 257-271) can use Null Object to represent the special case of not iterating over anything, and Adapters (Gamma *et al.*, 1993, pp. 139-150) can use Null Object so that they can pretend to adapt another object and yet not actually be adapting anything.

While all of these patterns could potentially be part of the CHECKS language, the language would then become overly complicated. At least some of the patterns would need to be removed in order to give the language back its focus.

### *Example (biological system):*

When embryonic stem cells specialize for particular tasks, such as to form a distinct organ, they need to be more separate from the mass of cells that were originally part of. They cannot develop a distinct identity while remaining fully connected to all the other cells around them. For example, when an embryo begins to form organs, a sac of fluid (fluid not being made up of cells but simply being a chemical solution) forms around the organ cells, creating a void in which the organ can develop.

# Chapter 6

# Idioms Language Analysis

This chapter presents an analysis of the C++ Idioms Pattern Language (Coplien, 2000a). The analysis uses the Language Designer's Pattern Language (LDPL) described in Chapter 5 to evaluate patterns in the idioms language and their interrelationships, and the overall structure of the idioms language. Section 6.1 provides an overview of how LDPL can be used to analyze and develop pattern languages, and Section 6.2 provides an overview of the idioms language. Sections 6.3 and 6.4 serve to demonstrate the kind of insights LDPL analysis offers. Section 6.3 highlights examples where LDPL analysis reaffirms the structure and place of a pattern or collection of patterns in the idioms language. In contrast, section 6.4 presents examples where using LDPL to analyze a pattern or collection of patterns leads to and guides suggested changes in the idioms language.

## 6.1 Using the Language Designer's Pattern Language

The Language Designer's Pattern Language (LDPL) can be used to analyze existing pattern languages and to evolve existing languages when forces dictate that they need to change. This chapter provides such an analysis of the C++ Idioms Language (Coplien, 2000a). At some point in the future, expert pattern language designers could use LDPL to develop pattern languages; this would be a further test of its usefulness and validity.

An important issue that needs to be clarified for LDPL to be used to analyze pattern languages concerns the process by which a language structure diagram is used to analyze, develop, and evolve existing languages. Some existing research (Alexander, 2002b, pp. 299-322; Porter *et al.*, 2005) suggests that sequences play an important role in pattern language application. While a detailed examination of sequences is beyond the scope of this thesis, Alexander describes a fundamental, overarching principle that governs the use of pattern languages, and the analysis described here uses LDPL in accordance with this principle. Alexander argues that patterns are applied one by one, in sequence, and that larger-scale patterns are used before smaller-scale patterns. The general process of using a language, then, is to start from the largest patterns and work gradually towards the smallest patterns. The process is not inflexible; for example, where one pattern is slightly larger in scale than another, the larger pattern is more likely to be better applied first, but it may be the other way around. Alexander (1975) also points out that not every pattern in a language will be used every time the language is used. The language structure diagram can be used to guide the sequencing of patterns; languages are structured from larger-scale down to smaller-scale patterns, and the language structure diagram reflects this organization, with an arrow from one pattern to another indicating that the former pattern is larger in scale than the latter. Porter and others (2005) examine the relationship between structure and sequencing in more detail.

The largest-scale pattern in LPDL is *Local Repair* and this governs, overall, the way LDPL is used in analysis of the idioms language. *Local Repair* states that when a change needs to be made to a language, it should be made locally but in a way that takes into account the impact on the whole system. If a new force needs to be incorporated into the language, then *Local Repair* notes that the force should be incorporated locally, within the pattern with which it best fits. The next largest patterns are *Local Symmetries* and *Cluster of Forces*. *Cluster of Forces* notes that each pattern draws together a group of related forces, and that the relationship between forces within a pattern is much stronger than that relationship of forces between patterns. *Local Symmetries* argues that smaller-scale patterns build on larger-scale ones by expressing the symmetry of larger-scale patterns more locally. The remaining patterns in LDPL focus on details of balancing forces and the interconnections between patterns; these remaining patterns are explained as they are used in analyzing the idioms language.

LDPL provides a way of analyzing the structural relationships between patterns shown by the arrows in a language structure diagram. Many pattern language designers use the arrows in a language structure diagram to indicate simply that one pattern builds on another, without indicating how or why. LDPL articulates key ways that patterns build on each other, and thus provides a means for assessing the validity of connections between patterns. For example, analysis of the idioms language with LDPL enables questions such as "How does `Handle/Body Hierarchy` build on `Concrete Data Type`?" to be answered by providing a means of analyzing the structures of `Handle/Body Hierarchy` and `Concrete Data Type`, and how structure is transformed when `Handle/Body Hierarchy` is added to `Concrete Data Type`. The patterns in the idioms language can also be analyzed to see how strongly the forces cluster within patterns, and the overall language structure can be studied to see if it can express necessary complexity.

## 6.2 Overview of the Idioms Language

The C++ Idioms language (Coplien, 2000a) draws on idioms from earlier work by Coplien (1992) to develop a pattern language that guides the building of an inheritance hierarchy in C++, with particular focus on algebraic types. Key issues in building such a hierarchy include efficient and effective memory management, placement of algebraic operations in the hierarchy, and implementation of operations that depend equally on more than one operand in a language without multiple dispatch.

Coplien provides a language structure diagram for his language, as shown in Figure 6-1, which attempts to articulate structural relationships between patterns in the language. An arrow from one pattern to another indicates that the pattern at the end of the arrow adds structure to, or builds on, the pattern at the beginning of the arrow. It is important to note that the language structure diagram is not a flowchart; the arrows indicate structural relationship, rather than order of application. There is, though, a connection between the structural position of a pattern in the language, and the order in which it is likely to be applied during a design process in which patterns are applied in sequence, as noted in Sections 6.1 and 3.2.2.

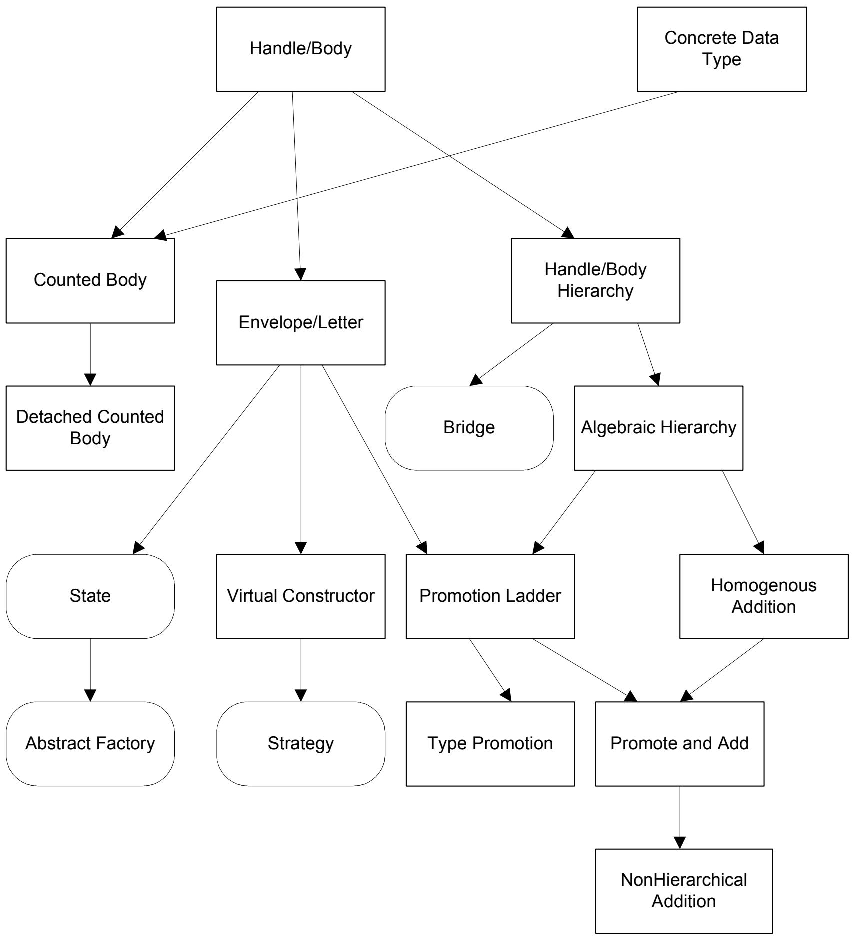

**Figure 6-1: A diagram of the language structure of the idioms language (Coplien, 2000a). An arrow from one pattern to another indicates that the latter pattern builds on the former. Patterns shown as rectangular boxes are part of the idioms language. Patterns shown as oval-like box shapes are drawn from the Gang of Four pattern catalog (Gamma *et al.*, 1995), but are included in the diagram to show how the idioms language potentially connects to other existing languages.**

The base patterns in the idioms language are Handle/Body and Concrete Data Type. All other patterns build on those two patterns, either directly or indirectly. Handle/Body splits a class into interface (handle) and implementation (body) classes, providing the convenience of scope-based memory allocation where the entity concerned is not in (program) scope. Concrete Data Type shows how to effectively manage memory allocation and de-allocation where object lifetime is dictated by external forces. Other patterns build on these base patterns to address further issues of memory management, becoming gradually more specific to an algebraic context. Counted Body builds on both Handle/Body and Concrete Data Type, providing reference counting for Handle/Body objects. Detached Counted Body builds on

Counted Body, showing how to do reference counting in a situation where body code cannot be modified. Envelope/Letter uses the inheritance mechanism to automate the duplication created by Handle/Body whereby changes to the handle class must be replicated in the body class. Virtual Constructor shows how to create Envelope/Letter objects without specifying the particular subtype of the Letter. Handle/Body Hierarchy creates Handle/Body classes in the context of an inheritance hierarchy. Algebraic Hierarchy outlines how to distribute algebraic operations in an inheritance hierarchy. The patterns below Algebraic Hierarchy in the language focus on the implementation of heterogeneous addition within such a hierarchy. The implementation approach used is to have an addition method in each subclass that deals only with operands of its own subclass (Homogeneous Addition), for each subclass to know how to promote itself to the next class up the hierarchy (Promotion Ladder), and then for operands of different type to be promoted to the same type before adding (Promote and Add, Non-Hierarchical Addition).

## 6.3 LDPL Analysis: Strengths in the Idioms Language

Key points that LDPL makes about pattern languages include that patterns build on other patterns by creating local symmetries, that smaller-scale patterns build on slightly larger-scale patterns, and that each pattern draws together and resolves a collection of related forces, where those forces arise out of the pattern's context. The examples in this section focus on patterns in the idioms language where analysis with LDPL reaffirms the structure of a pattern and its place in the idioms language.

### 6.3.1 Handle/Body to Envelope/Letter to Virtual Constructor: Forces, Symmetries and Scale

Handle/Body is a base pattern in the idioms language, as shown in Figure 6-1. It splits a design-time object into two linked objects, one of which acts as the object's interface (the Handle), and the other of which manages the object's implementation (the Body). The coordination of memory management for the object can then be encapsulated in the definition of a Handle class, making memory management less error prone, more consistent, more efficient, and more transparent. The Handle instance, which functions as a label for the Body entity, maintains a pointer to a body instance and manages the allocation and de-allocation of the body. The Handle simply forwards operations to the equivalent Body operation, which is where their implementation resides.

Handle/Body creates a situation where changes to the Handle class must be manually duplicated in the Body class, because Handle and Body share signature, and the role of the Handle is simply to forward operations to the Body. Maintaining Handle/Body is thus left in the hands of the programmer and potentially lacks coordination; this represents an unresolved force.

Envelope/Letter builds on Handle/Body and provides a way of coordinating the maintenance of Handle/Body objects. In addition to the Handle maintaining a pointer to the Body, Envelope/Letter makes the Body class inherit from the Handle class. The Body class thus obtains signature from the Handle class through the inheritance mechanism, and changes to the Handle class are propagated to the Body class by definition. Sometimes, though, it is desirable for a given Envelope/Letter object to be able to hold a Body (Letter) object from one of a collection of body classes, with the Handle (Envelope) able to be moved to point from one body class to another as appropriate at run time. Even at the point of object creation the client may not be able to specify the particular subclass of the Body object being instantiated, and, in fact, the client should not need to know about the implementation of the inheritance hierarchy; Envelope/Letter does not resolve this force.

Virtual Constructor addresses many forces left unresolved by Envelope/Letter. These include that the implementation of an inheritance hierarchy should be hidden from users of that hierarchy, that the method of selecting the appropriate derived class in a given situation should minimize coupling between all classes involved, and that the client needs to be able to use the operators of any derived class. Virtual Constructor thus clusters forces related to effective and efficient use of an inheritance hierarchy to create objects. The structure that it builds resolves these forces by showing how to build an inheritance hierarchy so that the particular derived class type required need not be specified by the client at compile time, but the client can still work with any derived type, or even with different derived types at different points in the program's execution. When Virtual Constructor is used, the client simply creates a new object of the Envelope class and sends whatever data it has to the Envelope constructor. The Envelope constructor is then responsible for deducing the particular subtype of the instance to be created.

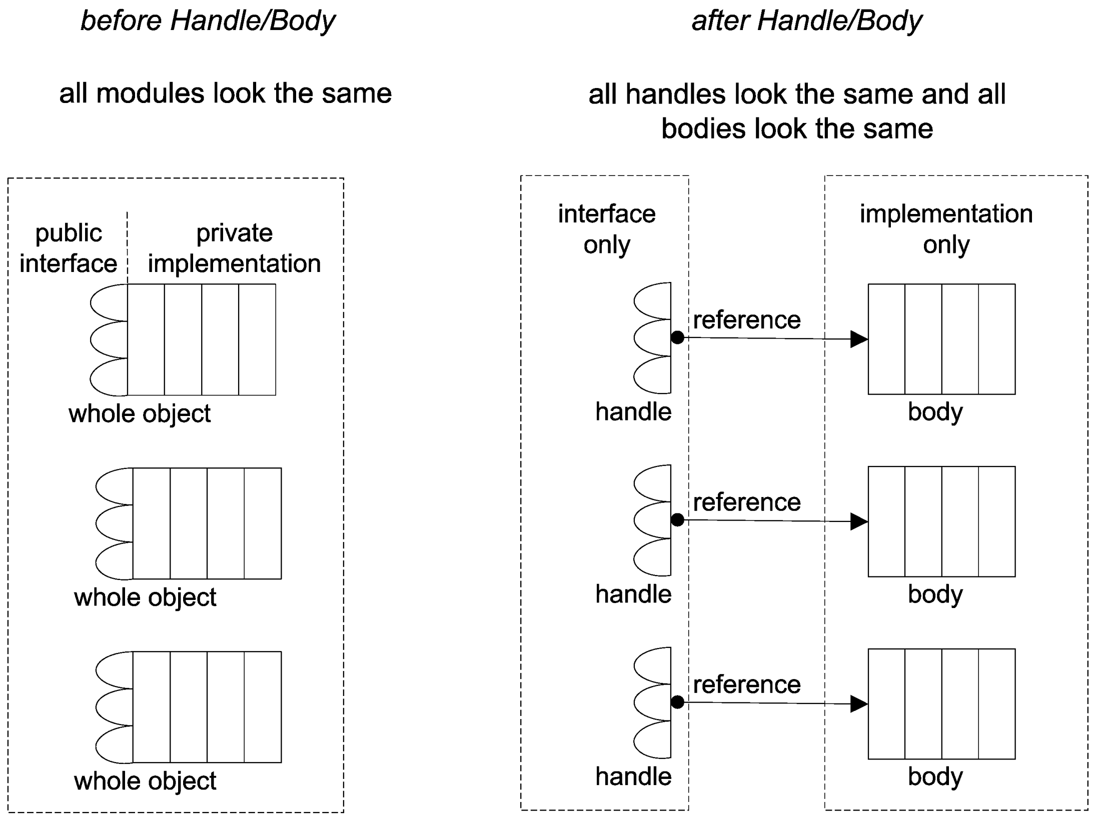

**Figure 6-2: Handle/Body transforms a design-time class into two distinct implementation classes, Handle and Body. In the process, the type symmetry of the original class is redistributed, with the Handle expressing the interface symmetry and the Body the implementation symmetry.**

LDPL can be used to analyze Handle/Body, Envelope/Letter and Virtual Constructor and the way they fit together. In particular, LDPL suggests that patterns build on other patterns by clustering and resolving forces (*Cluster of Forces*, *Resolution of Forces*); the forces addressed by Handle/Body, Envelope/Letter, and Virtual Constructor are discussed in the preceding paragraphs. But LDPL also suggests that patterns build on other patterns by creating structure that breaks symmetry (*Local Symmetries*) and that smaller-scale patterns build on larger-scale ones (*Levels of Scale*); these issues are addressed in the following paragraphs.

While Handle/Body does not build on any explicitly defined pattern, because it is a base pattern in the idioms language, it nevertheless creates local symmetries, as shown in Figure 6-2. The symmetry that Handle Body redistributes has to do with the original object's type. All objects of that type have the same interface and implementation; this is the invariant. What changes is the particular object under

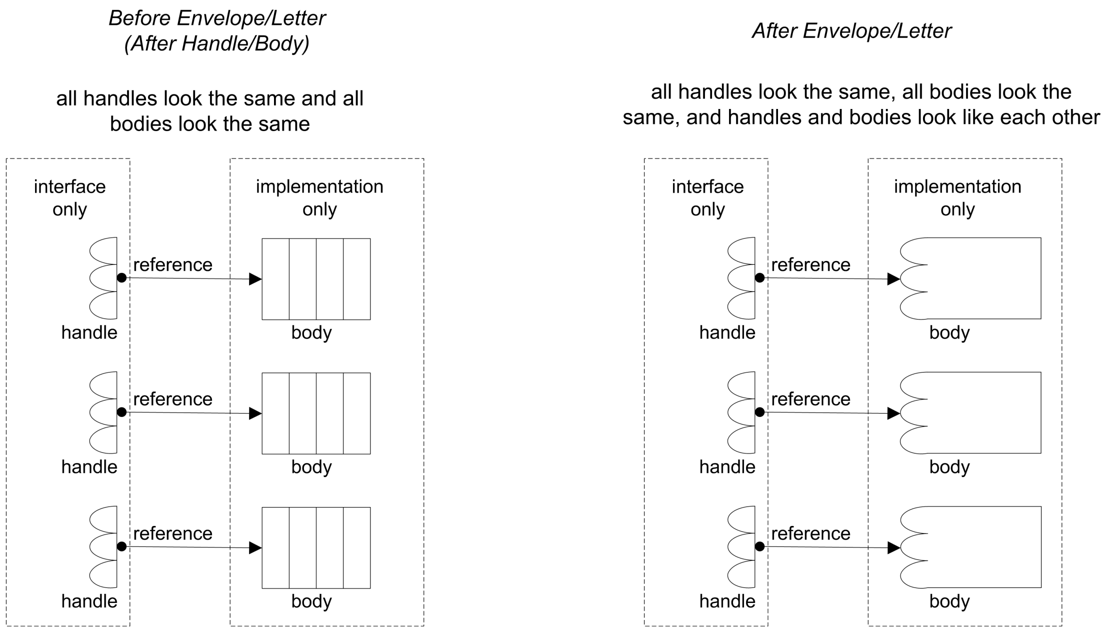
Before Envelope/Letter
(After Handle/Body)
all handles look the same and all bodies look the same
interface only
implementation only
reference
handle
body
After Envelope/Letter
all handles look the same, all bodies look the same, and handles and bodies look like each other

**Figure 6-3: Envelope/Letter makes the Body class inherit from the Handle class, avoiding unnecessary manual duplication of signature.**

consideration. Handle/Body preserves the symmetry, but redistributes it more locally. It splits the type symmetry into interface and implementation symmetry, with the interface symmetry expressed in the Handle object and the implementation symmetry in the Body object. The Handle and Body are each, though, still part of one, whole object: the Handle/Body object. The Handle/Body object thus still expresses full type symmetry, and the original symmetry is preserved, with Handle/Body creating local symmetries, as discussed in Chapter 5.

Envelope/Letter in turn builds on Handle/Body by creating local symmetries, as shown in Figure 6-3. Symmetry potentially exists in Handle/Body between Handle and Body classes; each is a different class, but shares the same signature. This symmetry is implicit; maintaining it is left in the hands of the programmer. Envelope/Letter makes this symmetry explicit. By deriving Body from Handle, and thus making Handle the base class for Handle/Body, Envelope/Letter creates an explicit symmetry between that base class and the Body class, or between Handle and Body classes. In creating this symmetry, Envelope/Letter resolves a force left unresolved by Handle/Body, namely that the maintenance of Handle/Body objects needs to be coordinated. Further, the symmetry created by Envelope/Letter is on a smaller level of scale than the symmetry created by Handle/Body; this symmetry

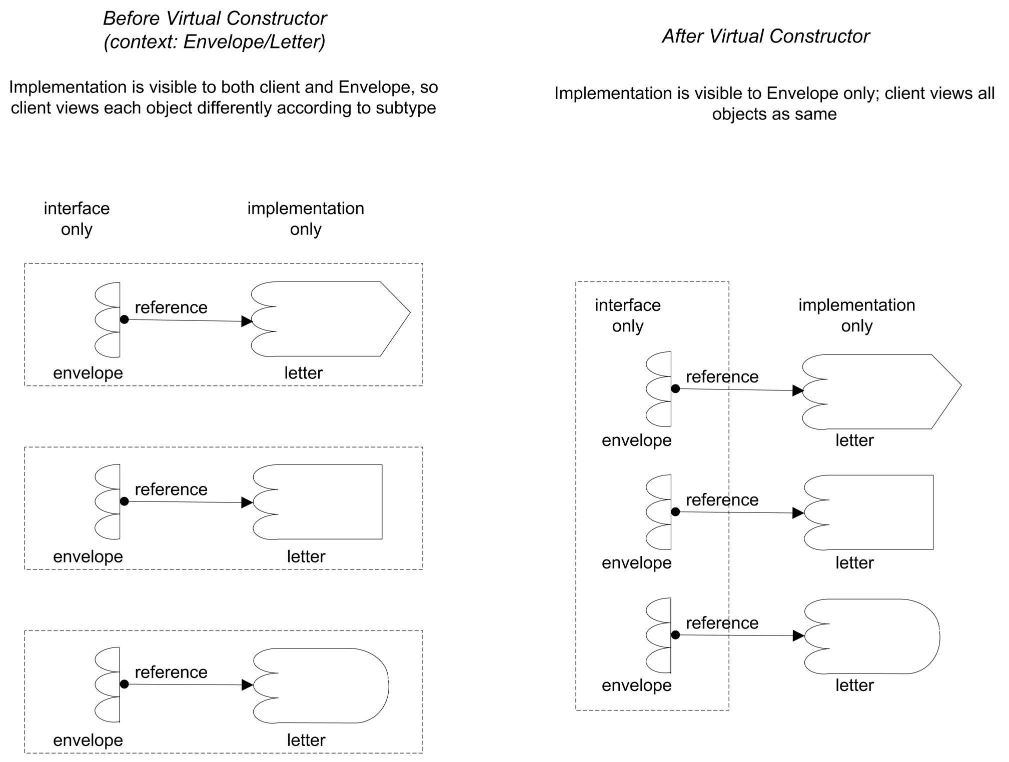
Before Virtual Constructor
(context: Envelope/Letter)
Implementation is visible to both client and Envelope, so client views each object differently according to subtype
interface only
implementation only
reference
envelope
letter
After Virtual Constructor
Implementation is visible to Envelope only; client views all objects as same
interface only
implementation only
reference
envelope
letter

**Figure 6-4: Symmetries created by Virtual Constructor. The client's view of objects is shown using dotted lines. Before Virtual Constructor is used, the client has a full view of Envelope/Letter objects and sees them differently. After Virtual Constructor, the client's view is restricted to the (identical) Envelopes.**

assumes the context of a Handle/Body object and the symmetries it creates, and works within those symmetries rather than changing them. It is for this reason that Envelope/Letter can be described as being smaller in scale than Handle/Body.

Virtual Constructor builds on Envelope/Letter by creating local symmetries. It allows an Envelope/Letter object to be created by the client in the same way regardless of the particular Letter subtype being created. What stays the same is the way the client creates the object, or the client's view of the object; what changes is the particular Letter subtype stored within the object. The symmetry created by Virtual Constructor is more local than that created by Envelope/Letter because it both works within an existing Envelope/Letter structure and applies to object creation rather than to an entire class structure. For this reason, Virtual Constructor is smaller in scale than Envelope/Letter. The symmetries created by Virtual Constructor are illustrated in Figure 6-4.

Analyzing Handle/Body, Envelope/Letter, and Virtual Constructor with LDPL thus confirms their positions in the idioms language. Envelope/Letter builds on Handle/Body by creating local symmetries, and is smaller in scale. Similarly Virtual Constructor builds on Envelope/Letter by creating local symmetries, and is slightly smaller in scale. Each pattern resolves a force or forces left unresolved by the pattern it builds on.

### 6.3.2 Counted Body: Cross Linkages Express Complexity

Counted Body builds on two patterns, Concrete Data Type and Handle/Body, as shown in Figure 6-1. Concrete Data Type is discussed in detail in Section 6.4.1. It provides guidelines about how to decide whether an object should be allocated on the stack or heap, and suggests that heap allocation should mirror stack allocation with the same object being responsible for both allocation and de-allocation of any given object. Handle/Body splits an existing entity into Handle and Body components, creating a context where many handles can potentially refer to one body instance.

The context for Counted Body is such that the programmer is responsible for the allocation/de-allocation of Handle/Body objects on the heap. Counted Body addresses the problem of memory management for Handle/Body objects, and shows how to manage multiple Handle instances for a given Body instance. The solution it provides is to add to each Body a count attribute that keeps track of the number of Handles referring to the Body, and to give the responsibility for appropriately incrementing the count – for doing the memory management – to the Handle.

LDPL notes that a pattern will both resolve forces and create new forces to be resolved by other patterns. In this example, Concrete Data Type creates a force to do with the memory management of objects on the heap. It identifies the importance of single object owners, but it does not address how to deal with situations where it is not possible or practical for a single object to own another object. Similarly, Handle/Body creates a force that the memory management of the allocation and de-allocation of bodies is in the programmer’s hands. It also creates the potential for unnecessary duplication of body objects.

Where single object ownership advocated by Concrete Data Type is not practical, corporate ownership, where several objects collectively act as the owner of another object, can still provide the benefits of Concrete Data Type. Likewise, allocation of body objects can be managed to provide the benefits of Handle/Body while also providing more efficient use of memory. There is no need for separate patterns to address the forces of the need for corporate ownership and the need to allocate bodies efficiently. These forces are in fact related and the location of Counted Body in the idioms language correctly indicates that Counted Body resolves them. LDPL recognizes that complex systems will give rise to situations like this, where different patterns of a similar level of scale give rise to related forces that should be resolved by a single pattern. LDPL pattern *Cross Linkages* notes that for a pattern language to be rich enough to generate complex systems, some patterns must break the symmetry of more than one larger-scale pattern.

Counted Body builds on Handle/Body by creating local symmetries. It splits the original Handle object into multiple Handle objects; conceptually all instances together form the one Handle for a given Body. From a symmetry point of view, in the transformation from Handle/Body to Counted Body, what stays the same is the client's view of the object, and how the client accesses it; a Handle/Body object is still viewed and accessed as one object, regardless of how its Handle is implemented. What changes is the particular Handle instance through which the Body is accessed at any given point; in Handle/Body there is only one instance, whereas in Counted Body there may be several, and access can be through any one of them, or through different Handles at different times.

Counted Body builds on Concrete Data Type by creating local symmetries in a similar way. Just as in Concrete Data Type, each object still has an owner (Handle), but in Counted Body that owner becomes composite. Instead of the owner of an object being one object instance, the owner of an object is made up of a collection of object instances. Those instances together act as the owner for an object. In effect, the first owner allocates the memory for the object in question, and the last remaining owner instance de-allocates it. All of the objects in the set act as owners for the same object and update the same count attribute; it is thus that they collectively act as the owner for the object in question. Local symmetries are created by the splitting of the original owner object into parts, each of which is itself an owner object instance. In the transformation from Concrete Data Type to Counted Body, what stays the same is that the object is allocated and de-allocated by an owner, and what changes is that the particular owner object instance that does the allocating is not necessarily the same as the one that does the de-allocating.

Figure 6-5 illustrates how Counted Body builds on a context created by both Handle/Body and Concrete Data Type to create local symmetries.

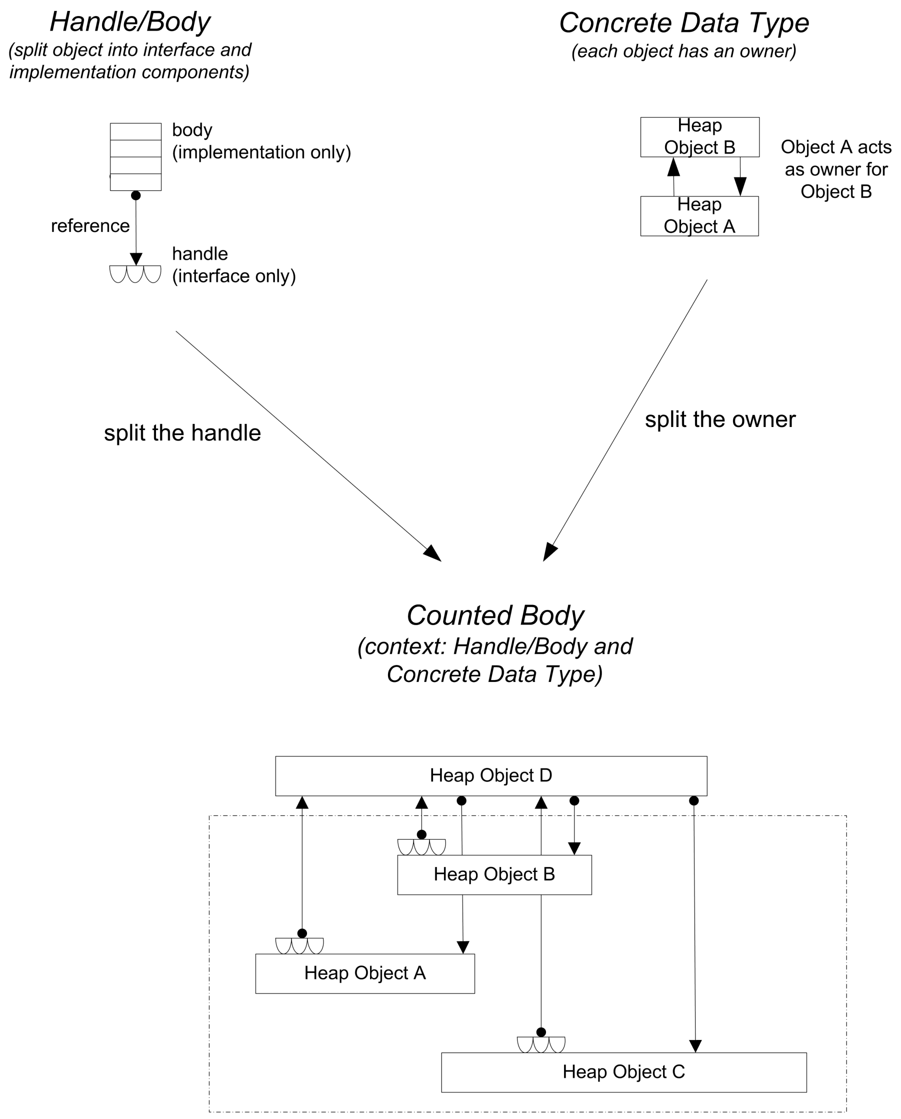

**Figure 6-5: Handle/Body creates local symmetries by splitting an object into Handle and Body components. Concrete Data Type creates local symmetries by mirroring stack allocation/deallocation symmetry on the heap. Counted Body builds on both Handle/Body and Concrete Data Type by creating local symmetries. Objects A, B, and C are all Handles for Object D and also collectively act as its owner.**

## 6.4 LDPL Analysis: Improving the Idioms Language

At some places in the idioms language, patterns do not work together in the way that LDPL suggests that they should. For example, a pattern may not resolve the forces it addresses, or it may not build on another pattern by creating local symmetries. Each of the following examples highlights a problem with the idioms language identified by using LDPL. LDPL is then also used to guide a suggested improvement to the idioms language.

### *6.4.1 Concrete Data Type: Solution Must Resolve Forces*

Concrete Data Type, shown in Figure 6-6, is one of the base patterns in the idioms language. It describes how to manage memory where variable lifetime does not necessarily follow program scope.

The context for Concrete Data Type is that of memory management in the procedural paradigm, and thus for built-in types in C++, where the underlying assumption is that the lifetime of a variable follows its program scope. Adding objects into this mix leads to designs in which the lifetime of a variable (object) ought not necessarily follow scope; Concrete Data Type addresses the problem of managing memory allocation in such situations.

Where lifetime follows program scope, memory for a variable can easily be allocated on the stack; at the point at which the enclosing block is added to the stack, the memory for the variable becomes available, and at the point at which the block is deallocated, the variable disappears. In contrast, heap allocation does not tie lifetime to program scope, so an object whose lifetime ought not follow program scope can be allocated on the heap; this is part of the solution proposed by Concrete Data Type.

A further problem, though, exists for the C++ programmer: the programming language cannot determine an object's lifetime from context. How, then, can the programmer decide whether or not an object's lifetime should follow its scope? And if lifetime does not follow scope, how should it be determined?

Concrete Data Type provides advice on these questions. It argues that objects that represent abstractions that "live 'inside' the program … should be declared as local (automatic or static) instances or as member instances" (Coplien, 2000) so that their lifetime or memory allocation does follow their scope. Such objects are thus either located on the stack or within the scope of another object. In contrast, objects that represent real-world entities should be allocated such that their lifetime follows their real-world lifetime. Such objects are allocated on the heap, using the language mechanism *new*. This is the solution proposed by Concrete Data Type, in its current form.

LPDL suggests that a pattern clusters related forces. While the question of how well Concrete Data Type clusters related forces can only be completely answered in the context of other patterns in the language and the kind of systems they build, there are two fundamental forces addressed by Concrete Data Type: first, that object lifetime need not follow program scope, and second, that if lifetime does not follow scope, it needs to be carefully managed to facilitate memory management. Each of the forces listed in the description of Concrete Data Type (Coplien, 2000a) elaborates on these two fundamental forces in some way.

LDPL also suggests that a pattern should resolve the forces it clusters. As it currently stands, the solution for Concrete Data Type states only that objects representing new or "real-world" entities "should be instantiated from the heap using the operator *new*" (Coplien, 2000a). Yet simply instantiating such objects from the heap does not guarantee the efficient and effective memory management Concrete Data Type is aiming for. While it is possible to make the lifetime of an object follow its real world lifetime just by allocating such objects on the heap, memory management in such a situation becomes a much more difficult problem, because it is dependent on a particular sequence of object construction and destruction. When an object construction and destruction sequence changes, the memory management will likely need to change. The force that an object's lifetime ought to be determined in a way that facilitates memory management is thus not necessarily resolved simply by allocating certain objects on the heap.

LDPL suggests that one way of seeing what the solution for Concrete Data Type ought to have been is to examine how Concrete Data Type breaks symmetry. LDPL states that patterns should build on other patterns by creating local symmetries; that is, expressing symmetry in those patterns more locally. Since Concrete Data Type is a base pattern in the idioms language, it does not build on any explicitly defined patterns. But the structure that it builds on is that defined by a stack-based approach to memory management. Analyzing symmetry in the stack-based approach reveals symmetry in the allocation and deallocation of memory blocks; the same block does both the allocation and deallocation for another block; what changes is whether memory is being allocated or deallocated. The symmetry is local in the sense that it applies to each particular block, rather than being just an overall symmetry of the whole stack, as shown in Figure 6-6. If Concrete Data Type builds on this structure, then it should locally reinforce this symmetry in some way.

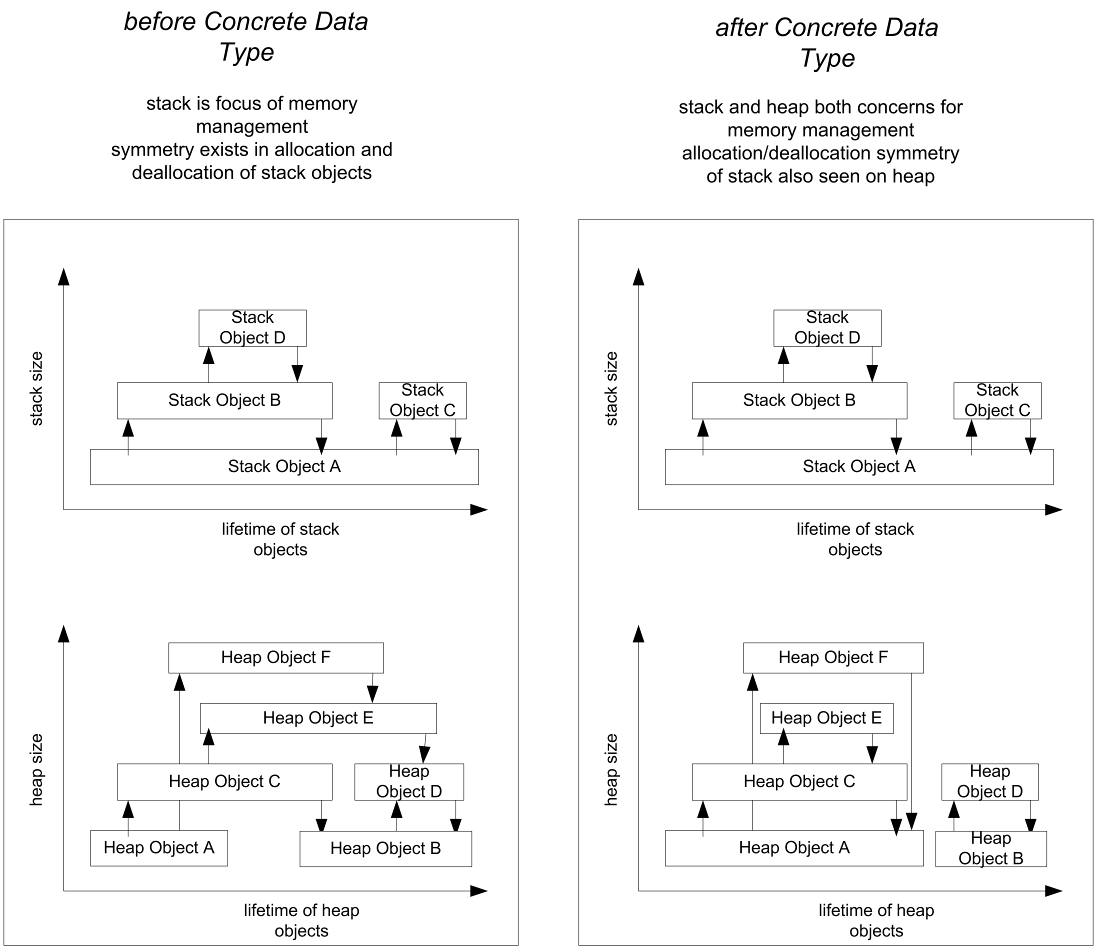
before Concrete Data Type
stack is focus of memory management
symmetry exists in allocation and deallocation of stack objects
after Concrete Data Type
stack and heap both concerns for memory management
allocation/deallocation symmetry of stack also seen on heap
stack size
Stack Object D
Stack Object B
Stack Object C
Stack Object A
lifetime of stack objects
heap size
Heap Object F
Heap Object E
Heap Object C
Heap Object D
Heap Object A
Heap Object B
lifetime of heap objects

**Figure 6-6: Use of Concrete Data Type ensures that each object has an owner; it is allocated and deallocated by the same object. This reproduces the symmetries of the stack-based approach to memory management. Note that symmetries apply to individual memory blocks (a small window of each picture), rather than to the stack or heap as a whole.**

Analyzing symmetry in the memory management approach suggested by Concrete Data Type reveals the same kind of symmetry as in the stack-based approach, *provided that the same object takes care of both allocation and deallocation for any one object.* In other words, each object should have an "owner". In the heap situation, what stays the same is which object does the allocation and deallocation for a particular object; what changes is whether the object is being allocated or deallocated. The symmetry is still local, but now in the sense that it applies to each particular object, rather than being an overall symmetry of the system of objects. The stack-based allocation/deallocation symmetry is then preserved for objects on the heap. And the symmetry of Concrete Data Type is *more* local than that which it builds on in the sense that the “world-view” now consists of both a stack and a heap, and each of those expresses local symmetries; the stack is now not the whole world-view, and its symmetries are now a smaller part of the world-view.

While the existing solution for Concrete Data Type may assume that each object has an owner, it does not explicitly say so. The assumption that the same object will manage both new and delete for any one object ought to be explicitly stated as part of the description of Concrete Data Type, because it is critical to the effective implementation of the pattern. If each object has an owner, then the force that memory management should be efficient and effective is resolved, because memory management can be efficiently and effectively incorporated within object constructor and destructor methods. Using LDPL to analyze Concrete Data Type enables its solution to be critiqued, and provides a way of working out how to improve the solution by analyzing symmetries in the pattern.

### 6.4.2 Detached Counted Body: Positioning Should Be Determined by Forces and Symmetries

Handle/Body creates the potential for many handles to refer to the same body. Managing the allocation and deallocation of bodies – reference counting – then becomes an issue. Both Counted Body and Detached Counted Body address reference counting for handle/body objects, but in different ways. Counted Body suggests modifying body code to include a count attribute that keeps track of the number of handles referring to a body. Detached Counted Body is appropriate to a context where body code cannot be modified. It provides reference counting by creating a separate count entity and having each handle associate itself with both a body and a count entity.

In the current language structure diagram for the idioms language, Detached Counted Body is shown as building on Counted Body. Yet, LDPL notes that each pattern clusters and resolves forces that arise out of the context in which the pattern exists, where that context is determined by the larger-scale patterns to which a pattern is connected. Detached Counted Body clusters forces related to memory management in a Handle/Body – not a Counted Body – context. For example, one of the key forces addressed by Detached Counted Body is that of there being potentially multiple handles for the one body. This is a force created by Handle/Body, not Counted Body. The other key force addressed by Detached Counted Body is that of the body code not being able to be modified. This force cannot be resolved by building on the solution

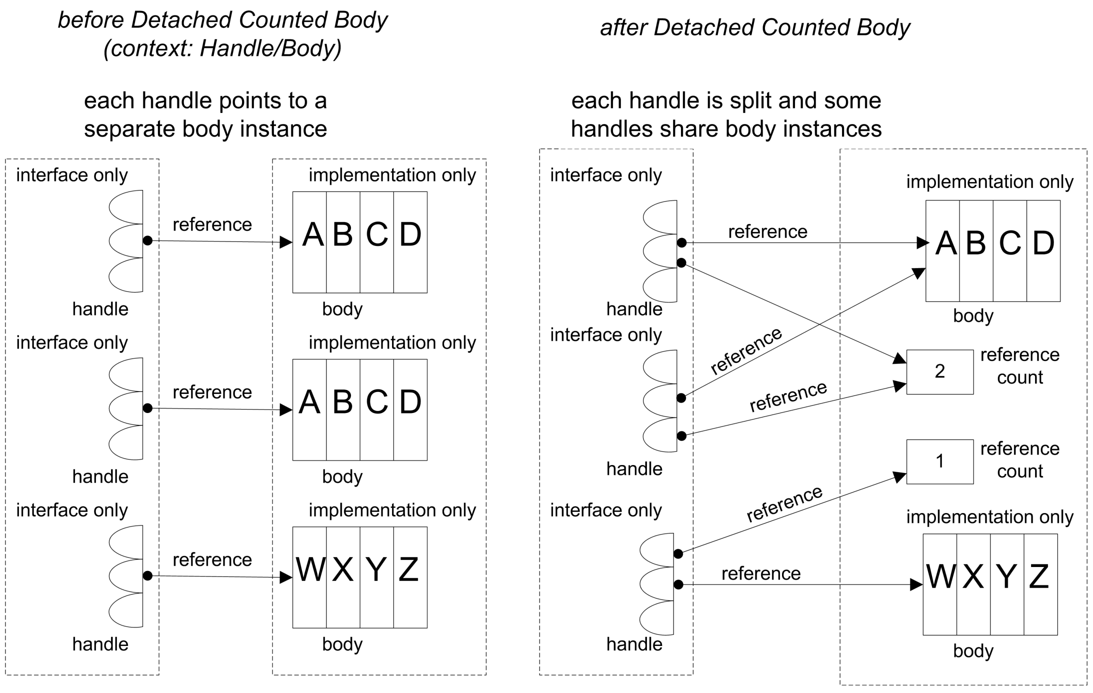
before Detached Counted Body
(context: Handle/Body)
each handle points to a separate body instance
after Detached Counted Body
each handle is split and some handles share body instances
interface only
implementation only
reference
handle
body
A B C D
W X Y Z
2
1
reference count

**Figure 6-7: Using Detached Counted Body avoids duplicating identical bodies and still facilitates memory management. Each handle splits into two pointers, one of which references a body and the other an object holding a count of the numbr of references to the body in question.**

of Counted Body, which requires that body code be able to be modified. In fact, both Counted Body and Detached Counted Body show how to implement reference counting. Each provides an implementation of reference counting for Handle/Body objects, appropriate to a slightly different context.

If Detached Counted Body does in fact build on HB then LDPL argues it should do so by creating local symmetries. As in Counted Body, Detached Counted Body splits an existing handle instance into separate handle instances that collectively form a Handle for a Body. In Detached Counted Body, however, since the Body instance cannot be modified, a count attribute is not added to the Body instance. Instead, a separate counting entity is created and each Handle instance associates count and Body entities by itself splitting in two and referencing both, as shown in Figure 6-7. There are thus two kinds of splitting taking place, and two kinds of local symmetry being formed.

One kind of local symmetry is the same as that in Counted Body; the one Handle instance is split into two (and potentially more) Handle instances, so that a collection of Handle instances together form the Handle for a Body. The other kind of local

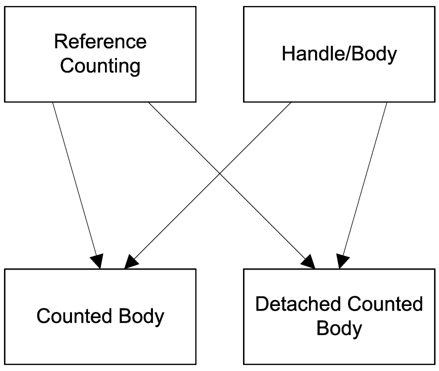

**Figure 6-8: Both Counted Body and Detached Counted Body build on Handle/Body but also address forces related to a fundamentally different idea, Reference Counting. It is therefore appropriate for both Counted Body and Detached Counted Body to build not only on Handle/Body but also on Reference Counting.**

symmetry being formed is within each Handle instance. Each Handle instance changes from being a single to a dual pointer. It is thus composed of two handle-like entities, one of which points to the Body and the other of which points to the count attribute for that Body. In this case, it is the client's view of a Handle instance that is invariant. What changes is which of the two reference pointers of the Handle is being accessed, depending on whether the count or other information about the Body is needed.

Analyzing the forces addressed by Counted Body and Detached Counted Body reveals that each of these patterns clusters and resolves forces that primarily have to do with reference counting. Yet neither Handle/Body nor Concrete Data Type - the patterns that Counted Body and Detached Counted Body build on according to the current language structure diagram - address reference counting. LDPL pattern *Common Ground* suggests that where patterns address similar clusters of forces and have a similar solution, they may well both build on a slightly larger-scale pattern. In the case of Counted Body and Detached Counted Body, such a slightly larger-scale pattern might be called Reference Counting. The existing language structure diagram for the idioms language, shown in Figure 6-1, does not incorporate a Reference Counting pattern and does not express the symmetry breaking relationship between Handle/Body and Detached Counted Body. Nor does it express the underlying commonality in Counted Body and Detached Counted Body. Figure 6-8 shows part of the language structure diagram that expresses those relationships.

### 6.4.3 *Algebraic Hierarchy: Each Pattern Should Add New Structure*

Coplien (2000a) argues that Algebraic Hierarchy is a distinct pattern that builds on Handle/Body Hierarchy[3], as shown in Figure 6-1, but describes Algebraic Hierarchy as though it is an application of Handle/Body Hierarchy in an algebraic context; that is, Algebraic Hierarchy is simply an example of Handle/Body Hierarchy. While the problem statements for the two patterns are different – Handle/Body Hierarchy mentions the need to be able to separate interface and implementation inheritance, and Algebraic Hierarchy asks how the inheritance hierarchy for algebraic types should be constructed – the forces and solutions for each pattern are essentially the same. The forces listed for both patterns all relate to separating interface and implementation inheritance, and the solution statement for Algebraic Hierarchy suggests building a Handle/Body Hierarchy (also known as a Bridge):

> "Use the Bridge pattern [Gamma+1995] to separate interface from implementation. The visible part of the Bridge is called class *Number*. It contains a pointer to a representation part, which contains the representation and operations of the specific type (*Complex*, *Real*, *Integer*, *Imaginary*, etc.)."
>
> (Coplien, 2000a)

If Algebraic Hierarchy is simply an application of Handle/Body Hierarchy then according to LDPL it is not a pattern in its own right; it is simply Handle/Body Hierarchy articulated in an algebraic context. If, on the other hand, Algebraic Hierarchy is a pattern in its own right and does build on Handle/Body Hierarchy, then LDPL argues that Algebraic Hierarchy must resolve a force or forces left unresolved by Handle/Body Hierarchy and the resolution of that forces or forces must add structure to Handle/Body Hierarchy by creating local symmetries. If Coplien's description of Algebraic Hierarchy is correct, then Algebraic Hierarchy does not add structure to Handle/Body Hierarchy and is redundant; if not, then the correct focus for Algebraic Hierarchy needs to be identified.

Handle/Body Hierarchy suggests that where classes have subtyping and implementation-sharing relationships that do not correspond with each other, interface and implementation inheritance can be split. In the context of classes that represent numeric types, this would enable, for example, class Real to be a subclass of Complex (as would be expected, since a Real is a Complex with an imaginary component of zero), while enabling class RealImplementation to inherit directly from

[3] Coplien depicts Algebraic Hierarchy as building on Concrete Data Type as well as Handle/Body Hierarchy, but the connection between Concrete Data Type and Algebraic Hierarchy is not the focus of this example.

NumberImplementation, rather than from ComplexImplementation. The class for Real numbers then gains the subtyping advantages that go with the *isA* relationship without carrying the space inefficiency of an unnecessary imaginary component. One advantage of subtyping that Handle/Body Hierarchy maintains is that classes share the same interface even if their implementation is different, so a method can be called without regard to the particular subclass. Another such advantage is that an object can change the type of its implementation (body) to another in the hierarchy without changing its interface (handle). So Handle/Body Hierarchy provides a structure that enables subtyping and implementation-sharing relationships to be different while important benefits of subtyping are maintained.

One of the unresolved forces that arises from Handle/Body Hierarchy is that when interface and implementation inheritance are split into separate hierarchies, some of the benefits of inheritance can be lost. For example, algebraic methods that depend on implementation data, such as *add* and *subtract*, must be explicitly redefined rather than inherited. This can result in designs that are inefficient and hard to maintain. In addition, algebraic methods like add and subtract are binary; they depend equally on two operands, so "dispatching on only one is unnatural" (Chambers, 1992). Implementing binary operations in a single-dispatch context is of itself a difficult problem; implementing such operations without inheritance is even more complex. Yet implementing basic algebraic operations is critical to an algebraic hierarchy because those operations embody fundamental operations necessary for the behavior of an algebraic type to be well defined.

Algebraic Hierarchy needs to address the problem of effectively implementing binary operations, without multiple dispatch, and in the context of a Handle/Body Hierarchy; that is, in a context where the usual inheritance mechanisms cannot be used. The solution it ought to provide is that the responsibility for algebraic operations be distributed throughout the hierarchy so that each subtype in the hierarchy is responsible for algebraic operations on parameters of its own type. When objects of differing types are involved, Algebraic Hierarchy arranges for a new object of the right type to be constructed and uses this object to explicitly call the right method. This solution scales better than many other solutions, which require changes to methods across all classes whenever a new class is added to the hierarchy. Instead, Algebraic Hierarchy only requires implementing the method in the new class being incorporated; methods in other classes remain unchanged.

LDPL argues that Algebraic Hierarchy should add structure to Handle/Body Hierarchy by creating local symmetries. If the solution structure for Algebraic Hierarchy is as described in the preceding paragraph, then the structure that Algebraic Hierarchy adds to Handle/Body Hierarchy is the algebraic methods, evenly distributed throughout the hierarchy. When a method is called on an object of one subtype, with a parameter of another subtype, Algebraic Hierarchy creates an object of the more general type from the more specific type. Instead of two objects of different types, the method need only operate on two objects of the same type. This is a symmetry, because methods can now assume that the called object and parameter are of the same type. The symmetry is local in the sense that is created within each method. A diagram of the symmetries is shown in Figure 6-9.

LDPL pattern *Local Repair* notes that where a change has been made to the structure of a pattern in a language, the patterns that build on that pattern, either directly or indirectly, also need to be analyzed to see how the change affects them. In the other examples of suggested improvements highlighted in Section 6.4, the patterns examined have either had no patterns underneath them or the change suggested has not modified the structure of the pattern, so further patterns have not been analyzed. In this case, the structure proposed for Algebraic Hierarchy is significantly different from the structure outlined in its original solution, and for this reason some of the patterns that build on Algebraic Hierarchy are further examined to see how they fit into the idioms language. The patterns selected for further analysis are Homogeneous Addition, Promotion Ladder, and Promote and Add, which are key to effectively implementing an Algebraic Hierarchy. While Type Promotion and Non-Homogeneous Addition also fall underneath Algebraic Hierarchy, they are less significant patterns that tie loose ends and so are not focused on here.

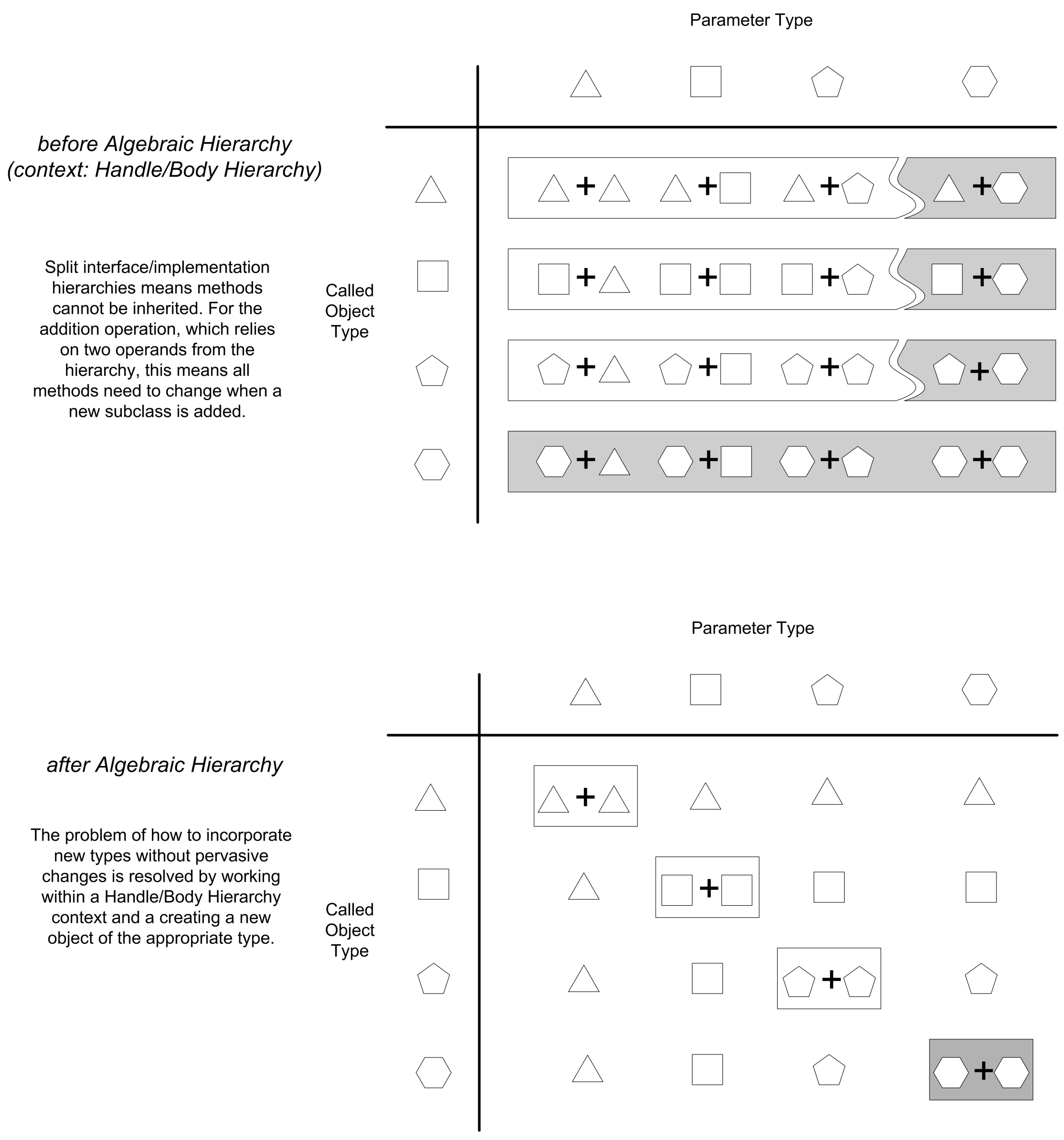
Parameter Type
before Algebraic Hierarchy
(context: Handle/Body Hierarchy)
Split interface/implementation hierarchies means methods cannot be inherited. For the addition operation, which relies on two operands from the hierarchy, this means all methods need to change when a new subclass is added.
Called Object Type
Parameter Type
after Algebraic Hierarchy
The problem of how to incorporate new types without pervasive changes is resolved by working within a Handle/Body Hierarchy context and a creating a new object of the appropriate type.
Called Object Type

**Figure 6-9: Local symmetries created by Algebraic Hierarchy. Object and parameter types are part of a class hierarchy and are represented as polygons; a greater number of sides indicates that the shape is further down the hierarchy. The picture is drawn as if the hexagon (six sides) is being added to an existing hierarchy. Boxes indicate method scope and shading indicates where method code needs to be changed to add the new class. Without Algebraic Hierarchy, addition of a new subclass requires widespread changes. Algebraic Hierarchy minimizes the scope of changes by creating a new object of the appropriate type when an object and its parameter type differ; the method itself needs only to manage an object and parameter of the same subtype. The symmetries exist in the creation of two objects of the same subtype from differing subtypes.**

In Coplien's original language structure diagram, Algebraic Hierarchy builds on Handle/Body Hierarchy. Promotion Ladder and Homogeneous Addition build on Algebraic Hierarchy, and Promote and Add builds on both Promotion Ladder and

Homogeneous Addition, as shown in Figure 6-10. Coplien describes Algebraic Hierarchy as a Handle/Body Hierarchy of algebraic types. He argues that Homogeneous Addition builds on this hierarchy by describing how to do addition for identical subtypes. Promotion Ladder builds on Coplien's Algebraic Hierarchy[4] by describing how to create classes that can promote themselves to their base class. Why a class should need to promote itself to its base class is not clear until you look at Coplien's Promote and Add, which builds on Promotion Ladder and Homogeneous Addition. Promote and Add uses a promotion ladder to do addition for differing subtypes by promoting them to the same subtype and then adding.

As Coplien describes them, Homogeneous Addition, Promotion Ladder, and Promote and Add each address one key aspect of how to implement binary operations within an algebraic hierarchy. The following paragraphs examine in more detail whether the patterns underneath Algebraic Hierarchy build on it's proposed new structure by resolving forces and creating local symmetries, as LDPL specifies that they should.

Homogeneous Addition is shown in Coplien's language structure diagram (Figure 6-1) as building on Algebraic Hierarchy. Coplien's version of Homogeneous Addition places *add* methods throughout an Algebraic Hierarchy such that each method only

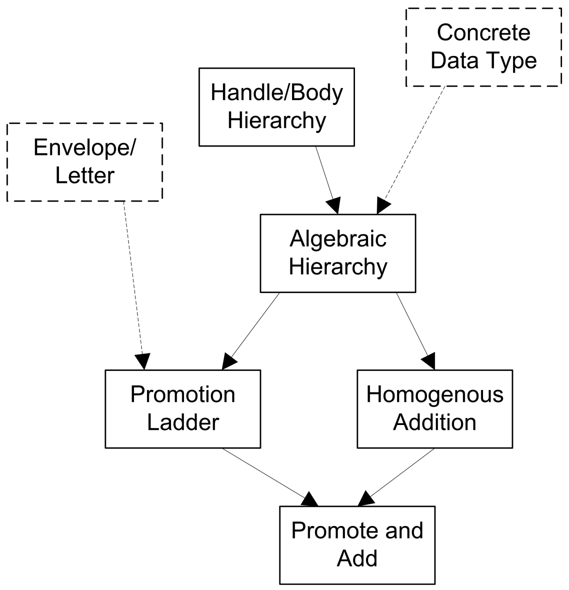

**Figure 6-10: part of the language structure diagram for the idioms language, showing Algebraic Hierarchy and surrounding patterns. Patterns outlined with a dotted line, while built on by Algebraic Hierarchy or related patterns, are not discussed in this section.**

deals with operands of its own subclass. If Homogeneous Addition builds on the new structure proposed for Algebraic Hierarchy, the structure of Homogeneous Addition needs to change slightly so that it is no longer distributing algebraic operations throughout the hierarchy; the new structure of Algebraic Hierarchy does that. The focus of Homogeneous Addition changes to how those methods are implemented. The new structure of Homogeneous Addition resolves a force left unresolved by the new solution structure of Algebraic Hierarchy, namely how to enable algebraic methods to create objects of the correct type for heterogeneous operations in a way that minimizes coupling and cohesion between classes, and does not require base classes to know about their derived classes.

Homogeneous Addition suggests that add methods should ignore the possibility of heterogeneous operations and simply add operands of the same type. This makes add methods cleaner and simpler because the programmer does not have to specify which add method should be called, according to the subtype of the argument. Instead, the add methods are able to function as any other method does, as if they only depend on one argument. This makes the dispatch of the add method symmetric despite the single dispatch context within which it is implemented, because the same method is used regardless of which argument is used for the dispatch. The dispatch of each add method is thus a local symmetry, and Homogeneous Addition builds on Algebraic Hierarchy by creating local symmetries.

Promotion Ladder is also shown in Coplien's language structure diagram as building on Algebraic Hierarchy. Promotion Ladder describes how to enable a class to promote itself to the next class up a hierarchy. It suggests that the knowledge about type promotion should reside in the derived class, with each class knowing how to promote itself to its base class. It is not clear, however, what the connection between either version of Algebraic Hierarchy and Promotion Ladder is. In LDPL terms, Promotion Ladder does not resolve a force left unresolved by either Coplien's Algebraic Hierarchy or the new proposed solution structure for Algebraic Hierarchy.

Even though Coplien places Promotion Ladder below Algebraic Hierarchy in his language structure diagram, in the description of the pattern he suggests Promote and Add actually provides the context for Promotion Ladder (Coplien, 2000a, p. 192). He also suggests that Algebraic Hierarchy is part of the context for Promote and Add (Coplien, 2000a, p. 190). Promote and Add shows how to do heterogeneous addition by promoting the more specific of two types to the more general type, and then using homogeneous addition to add the resulting two numbers together. By enabling numbers of different type to be added without algebraic methods needing to have knowledge of type promotion, Promote and Add resolves a force left unresolved by the new solution structure of Algebraic Hierarchy, namely that changes to the class hierarchy itself require pervasive changes to algebraic methods. Promote and Add acknowledges the need for type promotion, but separates type promotion from the algebraic methods. Promote and Add, together with Homogeneous Addition, builds on Algebraic Hierarchy to provide a more effective and efficient way of doing addition within a Handle/Body Hierarchy than that provided by Algebraic Hierarchy. Promote and Add creates local symmetries by allowing two numbers of different subtypes to be treated as if they are of the same subtype.

Promotion Ladder can now fit into the idioms language underneath Promote and Add by resolving unresolved forces. While Promote and Add acknowledges the need for type promotion, it does not resolve forces to do with how type promotion should be done or where the knowledge of type promotion should go. Promotion Ladder resolves these forces by suggesting that each class need only know how to promote itself to its own base type, with promotions involving more than two levels of a class hierarchy being handled using multiple successive promotions, hence the term Promotion Ladder. Coplien notes that Promotion Ladder "retains good coupling and cohesion properties; base classes needn't know about their derived types. The work necessary to add a new type, and knowledge of how it should be promoted to other types, is kept to a minimum." Promotion Ladder builds on Promote and Add by creating local symmetries because the way promotion is done is the same in every subclass, and the knowledge of type promotion is distributed throughout the hierarchy.

The LPDL analysis of Algebraic Hierarchy, Homogeneous Addition, Promotion Ladder, and Promote and Add demonstrates that the patterns underneath Algebraic Hierarchy can build on its proposed new solution structure by creating local symmetries that resolve unresolved forces. It also suggests that the positions of Promotion Ladder and Promote and Add in Coplien's original language structure diagram need to be swapped, with Promotion Ladder building only on Promote and Add and not on Homogeneous Addition. This leads to a proposed new language structure diagram for that part of the idioms language shown in Figure 6-10; the new diagram is shown in Figure 6-11.

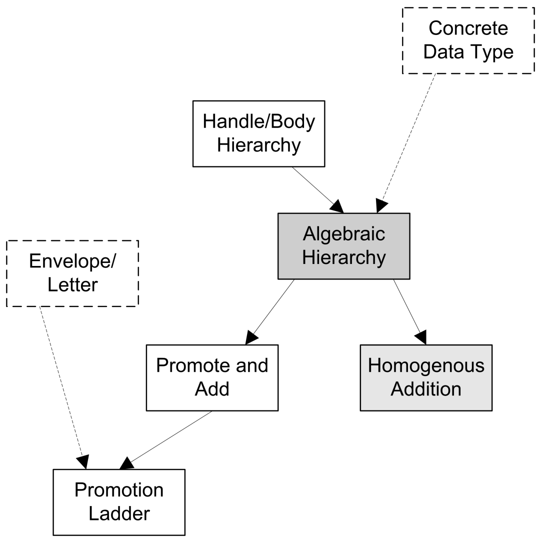

**Figure 6-11: new language structure diagram for Algebraic Hierarchy and surrounding patterns. Patterns outlined with a dotted line have not been discussed in this section. Algebraic Hierarchy is shown shaded to indicate that its structure is substantially different from Coplien's original pattern. Homogeneous Addition is shown lightly shaded as it has undergone a small structural change. Promote and Add and Promotion Ladder have swapped places, with Promotion Ladder building only on Promote and Add.**

# Chapter 7

# Conclusions and Future Directions

The research goals of this project were as follows:

- To better define the pattern concept.
- To better define the way patterns work together in a pattern language.

This thesis has addressed these goals by focusing on the structure of patterns and pattern languages, and the processes by which they are built. The theoretical foundation of the work is existing theory on patterns, pattern languages and symmetry breaking. The form of the work is itself a pattern language that articulates the structure of pattern languages and the key processes by which they form and evolve, and thus guides the building of a properly structured pattern language. The language uses multidisciplinary examples to validate the claims made, and an existing software pattern language is analyzed using the material developed.

Section 7.1 revisits the contributions listed in Chapter 1 and summarizes how those contributions have been achieved. Section 7.2 focuses on the implications of those contributions within the broader context of patterns research. In Section 7.3, two significant conclusions arising from this work are presented. Section 7.4 outlines possible areas for future research and Section 7.5 provides brief concluding remarks.

## 7.1 Contributions

This thesis makes the following contributions to pattern-based research:

## *It establishes foundations for a generative semantic model for pattern languages*

The core of this thesis describes the Language Designer's Pattern Language (LDPL), a language that itself articulates how patterns in a language work together. LDPL was developed using existing theory on symmetry breaking, pattern languages, and complex systems. Many existing patterns, most of which came from one of three pattern languages, were analyzed as part of the process of developing LDPL, and the analysis shows that their relationships with other patterns can be understood as described by LDPL.

Although other work developed prior to LDPL has contributed to the foundations for a generative semantic model for pattern languages, such efforts have generally been limited in focus or scope. Meszaros and Doble's (1998) language is generative, but its focus is largely on syntax rather than semantics. Similarly Noble and Biddle's (2002) work is limited to an object-oriented software context and Buschmann *et al.* (1996) focus on the particular domain of software architecture and do not fully analyze the connections between patterns at different levels of scale. Alexander's (1977; 1979) work is clearly foundational to the patterns field, and he offers numerous insights on how to build a pattern languages that are incorporated in LDPL. However, he does not attempt to define a model.

## *It articulates the structure of pattern languages, and the processes by which they are built*

This thesis recognizes that a pattern is fundamentally both a "process and a thing" (Alexander, 1979, p. 247); it is both a transformation on system structure and the structural configuration resulting from that transformation. For this reason, articulating how patterns work together requires expressing their structures and the processes by which they transform those structures. The thesis describes the structures of patterns in terms of local symmetries and the processes by which they transform structure in terms of small-step, local change driven by global perspective.

## *It uses a pattern language to express the structure of pattern languages, and the processes by which they are built*

LDPL expresses the structure of pattern languages and the processes by which they are built. Each pattern in LDPL describes one key aspect of building a pattern language. *Local Symmetries*, for example, notes that smaller-scale patterns transform larger-scale patterns by creating structure that reinforces existing symmetry more locally. The pattern incorporates both the transformation and the resulting structure. LDPL as a whole, being the patterns and their interconnections, governs the kind of overall structures and process being described.

### *It presents a pattern language that can be used to analyze and improve existing languages and potentially to help build pattern languages*

LDPL was used to analyze and suggest improvements to the C++ idioms language. Part of this analysis involved rebuilding small parts of the idioms language. Further experimentation involving the analysis of a language not used as a source of examples for LDPL would increase the strength of this claim. Further experimentation where expert pattern language designers use LDPL to develop pattern languages, as described in Section 7.4, is required to more thoroughly demonstrate the capacity of LDPL for building languages.

### *It uses symmetry-breaking theory to analyze pattern languages and systems*

This thesis builds on existing analyses of patterns in terms of symmetry breaking by explicitly bringing symmetry-breaking analysis into a pattern *language* context. It demonstrates that symmetry breaking is not just a means of analysing isolated patterns but a foundation, together with concepts like forces, scale, and local repair, for describing and analysing pattern languages.

### *It defines a precise relationship between patterns of different levels of scale*

Many authors use arrows in a pattern language diagram to indicate relationships between patterns. Some authors (Bradac and Fletcher, 1998; Molin and Ohlsson, 1998) have used arrows in a pattern language diagram without precisely defining what they mean. Others (Gamma *et al.*, 1995; Keller and Coldewey, 1998) have used arrows to indicate a wide variety of relationships between patterns. This thesis demonstrates that the arrows in a pattern language diagram can be defined in symmetry-breaking terms; an arrow from one pattern to another indicates that the smaller-scale pattern breaks the symmetry of the larger-scale pattern.

***It draws together theory on the nature of order, symmetry breaking and pattern languages***

This thesis combines insights from a number of different research areas into a coherent whole. Significant research areas drawn on include Alexander's early work on pattern languages, his much later work on the nature of order, with its symmetry-breaking themes, work on symmetry breaking in natural, physical systems, and a collection of other research in areas like design, software design, and complex systems theory. Many of the areas of research LDPL builds on were not previously noted as being significantly connected or the connections between areas were not well researched.

***It identifies common threads in discussion of patterns from a wide variety of domains and used these to better define the pattern concept***

This thesis examines the use of the term "pattern" in domains as broad as software, architecture, natural physical systems, organizational systems, and psychology. It builds an understanding of patterns coherent with both colloquial use and more technical use, in each of the domains examined.

## 7.2 Implications for Patterns Research

This thesis sits in the context of several other major pieces of work. In particular, as discussed in Section 3.2.2, Meszaros and Doble (1998) have previously developed a pattern language for writers of pattern languages. While Alexander (1977; 1979) did not write a pattern language about how to write pattern languages, he does offer much insight into the task in his book *The Timeless Way* (1979). As discussed in Section 4.3.3, Coplien and Zhao (2001; 2003) are engaged in ongoing research into the use of symmetry as a formalism for expressing fundamental pattern and pattern language concepts.

LDPL could be used in conjunction with the language that Meszaros and Doble describe. Their language provides a mix of basic and sophisticated information at a conversational, beginner level, helping to demystify the process of writing a pattern language. It is, however, quite possible to write something using Meszaros and Doble's language that looks, feels, tastes and smells like a pattern – but is not actually a pattern. By not focusing on the deep, underlying, and complex structural relationships at the heart of the pattern concept, Meszaros and Doble risk fostering the development of "pattern" languages that are not generative or structural, and lack many of the qualities described by Alexander (1977; 1979) as essential. LDPL, on the other hand, requires a more sophisticated understanding of pattern in order to be useful, but once that understanding is there, conscientious use of LDPL is much more likely to result in real patterns, with the designer's understanding of patterns and pattern languages potentially improved through the process. The two languages could potentially complement each other well.

Alexander (1977; 1979) offers many insights into how to build a pattern language, even though he does not articulate those insights in the form of a pattern language. LDPL taps into many of Alexander's insights, such as that smaller patterns build on larger ones through a process of differentiation, and that patterns both resolve forces and generate new forces to be resolved by smaller-scale patterns. The development of LDPL validates many of Alexander's insights in the sense that it examines the plausibility of those insights in terms of existing pattern languages and their structure, and what can be gleaned from their structure about the process of developing those languages. LDPL draws many of Alexander's insights together into a tangible entity that constitutes a model for building pattern languages.

Coplien and Zhao (2001; 2003) have been the major instigators of the push to investigate the relationship between symmetry, symmetry breaking, and software patterns. Their focus is at a much more formal level than that of this thesis; they are seeking to formalize the relationship between symmetry breaking and software patterns. LDPL does not seek to formalize this relationship, or indeed even to formalize patterns. Rather, it uses symmetry breaking at an informal level as a useful concept for understanding patterns and how they work together. By demonstrating the relevance of symmetry at an informal level, the development of LDPL adds weight to the push to further investigate the connections between symmetry and pattern at a more formal level, not because the pattern concept can ever be completely defined by formalizing it, but because those using and writing patterns and pattern languages still need to develop their understanding of patterns at a theoretical as well as practical level in order to identify and use them well.

## 7.3 Conclusions

Two conclusions from this work stand out as particularly significant:

### *It is appropriate to develop a pattern language for building pattern languages*

This thesis worked from the assumption that a pattern language is a complex, designed system, and that therefore, as for other such systems, it makes sense to use a pattern language to describe, generate, and analyze pattern languages. The assumption is demonstrated to be appropriate by both the wide variety of patterns able to be analyzed in LDPL and the success in using LDPL to analyze the idioms language. Given that success, it is likely that pattern language designers across a wide variety of fields will be able to use LDPL in developing and analyzing pattern languages, and that system designers will be able to use pattern languages to describe and analyze their systems, both statically and dynamically as those systems are transformed over time.

### *Pattern language designers can benefit by understanding symmetry breaking in natural, physical systems*

This thesis worked from the assumption that analysis of evolved systems offers useful insights to system designers. In particular, the symmetry-breaking processes taking place in natural, physical systems lead to the formation of patterns in those systems, and so it is worthwhile for pattern language designers to understand those processes. An understanding of those processes was critical to the development of LDPL.

LDPL defines patterns and the connections between them in terms of the creation of local symmetries. Such a definition requires an acknowledgement of the importance of scale; smaller-scale patterns break the symmetry of larger-scale patterns. It also requires an acknowledgement of the importance of forces. It is forces that dictate the need for structure to change, and define the parameters for the structure of a pattern; the symmetry breaking must generate structure that holds a set of forces in tension. Even in domains that do not appear to be geometric, such as software, forces still drive the need for change. This thesis has analyzed symmetry breaking in software in several instances. It is likely that other software structure can be understood in geometric terms by noting symmetries that are transformed through the application of a pattern.

## 7.4 Future Work

This section lists and elaborates on significant areas of potential further research that arise from this thesis.

### *Exploring pattern language sequences*

Alexander (1977, pp. xii-xiii) argues that the order in which patterns are applied is critical to the effectiveness of the language. At a basic level, ordering is specified with respect to the level of scale of patterns in a language; larger-scale patterns are applied before smaller-scale patterns, and patterns of a similar level of scale are applied as closely together as possible. A selection of patterns from a language, applied one by one from larger to smaller scale, is referred to as a *sequence*. In more recent work (Alexander, 2002b, pp. 299-322), Alexander argues that good sequences help the designer create a coherent whole at each step, and that of all the possible sequences for any given language, only a few will be workable. One avenue of future research would be to identify those LDPL sequences that ought to predominate and to understand why those sequences are worth using whereas others are not. Another avenue would be to examine the connection between structure and process in a pattern language and see if the connection could be more precisely defined using sequences.

### *Using LDPL to develop pattern languages*

One of the important tests of the validity of LDPL will be to compare its use in the development of pattern languages with what expert pattern language developers do anyway. One way of testing this would be for expert pattern language developers to use LDPL to develop pattern languages. The developers would need to provide feedback on differences and similarities between their usual practice and the LDPL approach, and whether or not they find LDPL useful, and why.

### *Analyzing patterns in LDPL using LDPL*

Since LDPL is a pattern language that aims to articulate the structure of pattern languages and the processes by which they are built, LDPL ought to be able to be analyzed according to its own principles. For example, *Cluster of Forces* could be analyzed to see whether it builds on *Local Repair* by creating local symmetries, or *Local Symmetries* could be analyzed to see what forces it clusters and resolves. This kind of analysis is another important way of furthering validating LDPL.

### *Using LDPL to analyze several more pattern languages in detail*

Analysis similar to that done on the idioms language could be done on other languages. In particular, such analysis could be done on Coplien and Harrison's (2004) substantial organizational pattern languages; the substantial nature of these languages together with the interesting nature of the context they have come out of – developing organizational structures that are conducive to successful software development – make them a particularly worthwhile area for further investigation.

### *Exploring the connection between Grenander, Alexander, and symmetry breaking*

Grenander (1996) has developed a theory of patterns that is grounded in the mathematics of group and set theory, and concepts of symmetry. While his theory is more focused on pattern recognition, rather than pattern generation, his work may offer insights useful to others seeking to define patterns in terms of symmetry. A more precise understanding of the relationship between what Alexander and Grenander's respective concepts of pattern may also offer insight useful to those working with patterns and pattern languages, and to those (Coplien and Zhao, 2001) attempting to formalize the connection between symmetry and Alexandrian patterns in some way.

### *Analyzing Alexander's language in terms of LDPL*

There are many ways in which Alexander's language could be analyzed to see how well his patterns and their interconnections fit with the kind of patterns and interconnections stipulated in LDPL. For example, Alexander (1977, pp. xiii-xv) marks his patterns with one, two, or three stars according to his degree of confidence in the pattern. It would be interesting to explore whether those patterns that clearly break the symmetry of other patterns are the ones rated with three stars. In terms of the overall structure of Alexander's language, Alexander does not provide a language structure diagram of his whole language, but does note which patterns are "related to" other patterns. It would be interesting to explore whether the connections that Alexander highlights between patterns always reflect symmetry-breaking relationships.

### *Analyzing symmetry connections across domains*

Some experts that have worked in more than one domain may have noticed common patterns across domains. For example, a biologist turned lace-maker mentioned in conversation that the patterns she used to see in biology she now sees when making lace. An interesting research question is the degree to which various patterns are "universal" or apply across domains. One starting point for exploring this question would be talking to experts who have worked in at least two different domains and identified common patterns across those domains.

#### ***Developing characteristics of pattern languages***

Some people beginning to explore pattern languages need to start at a more basic level than that offered by LDPL. Earlier work (Winn and Calder, 2002) saw patterns defined in terms of characteristics; equally, there is a place for defining pattern languages in terms of characteristics.

## 7.5 Concluding Remarks

This thesis has opened up a new area of research by exploring the use of pattern languages as a way of representing the structure and transformation of complex systems. Researchers in many fields, and particularly those working with complex systems in any domain, can potentially draw significant benefit from the research

# Bibliography

Alexander, C. (1964) *Notes on the Synthesis of Form,* Harvard University Press, Cambridge, MA, USA.

Alexander, C. (1975) *The Oregon Experiment,* Oxford University Press, New York, New York, USA.

Alexander, C. (1979) *The Timeless Way of Building,* Oxford University Press, New York.

Alexander, C. (1988) A City is not a Tree. In *Design After Modernism: Beyond the Object* (Ed, Thackara, J.), Thames and Hudson, London, UK, pp. 67-84.

Alexander, C. (2002a) *The Nature of Order (Book One): The Phenomenon of Life,* The Center for Environmental Structure, Berkeley, California, USA.

Alexander, C. (2002b) *The Nature of Order (Book Two): The Process of Creating Life,* The Center for Environmental Structure, Berkeley, California, USA.

Alexander, C., *et al.* (1977) *A Pattern Language: Towns, Buildings, Construction,* Oxford University Press, New York, New York, USA.

Altshuller, G. (1984) *Creativity as an Exact Science: the Theory of the Solution of Inventive Problems,* Gordon and Breach, New York, New York, USA.

Angluin, D. (1980) Inductive Inference of Formal Languages from Positive Data. In *Information and Control,* Vol. 45, pp. 117-135.

Anthony, D. L. G. (1996) Patterns for Classroom Education. In *Pattern Languages of Program Design 2* (Eds, Vlissides, J. M., *et al.*), Addison-Wesley, Reading, Massachusetts, USA, pp. 391-406.

Appleton, B. (2000) *Patterns and Software: Essential Concepts and Terminology,* <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>, accessed March 17, 2003.

Bar-Yam, Y. (1997) *Dynamics of Complex Systems,* Addison-Wesley, Reading, Massachusetts, USA.

Bar-Yam, Y. (2000-2005) *About Complex Systems,* <http://necsi.org/projects/yaneer/points.html>, accessed August 4, 2004.

Beck, K. (2000) *Extreme Programing: Embrace Change,* Addison-Wesley, Reading, Massachusetts, USA.

Beck, K., *et al.* (2001) *Manifesto for Agile Software Development,* <www.agilemanifesto.org>, accessed July 1, 2003.

Berczuk, S. (1995) Book Review: Object Models: Strategies, Patterns, and Applications. In *Object Oriented Systems,* Vol. 2, pp. 190-192.

Bergin, J. (2000) Fourteen Pedagogical Patterns. In *Proceedings of the Fifth European Conference on Pattern Languages of Programs (EuroPLoP)*, pp. unknown. Conference in Irsee, Germany.

Bergin, J. (2002) *Some Pedagogical Patterns*, <http://csis.pace.edu/~bergin/patterns/fewpedpats.html>, accessed May 29, 2003.

Bern, E. (1964) *Games People Play,* Penguin Books, New York, New York, USA.

Blundell, T. L. and Srinivasan, N. (1996) Symmetry, stability, and dynamics of multidomain and multicomponent protein systems. In *Proceedings of the National Academy of Sciences USA,* Vol. 93, pp. 14243-14248.

Bradac, M. and Fletcher, B. (1998) A Pattern Language for Developing Form Style Windows. In *Pattern Languages of Program Design 3* (Eds, Martin, R.*, et al.*), Addison-Wesley, Reading, Massachusetts, USA, pp. 347-357.

Brand, S. (1997) *How Buildings Learn: What Happens After They're Built,* Phoenix Illustrated, London, UK.

Bruegge, B. and Dutoit, A. H. (2000) *Object-Oriented Software Engineering: Conquering Complex and Changing Systems,* Prentice Hall, Upper Saddle River, New Jersey, USA.

Buschmann, F. and Meunier, R. (1995) A System of Patterns. In *Pattern Languages of Program Design* (Eds, Coplien, J. O. and Schmidt, D. C.), Addison-Wesley, Reading, Massachusetts, USA, pp. 325-343.

Buschmann, F.*, et al.* (1996) *Pattern-Oriented Software Architecture: A System of Patterns,* John Wiley & Sons, Chichester, England.

Cambridge University Press (2003) *Cambridge Dictionaries Online*, <http://dictionary.cambridge.org/>, accessed May 20, 2003.

Campbell, N. A. (1996) *Biology,* Benjamin/Cummings, Menlo Park, California, USA.

Chambers, C. (1992) Object-Oriented Multi-Methods in Cecil. In *Lecture Notes in Computer Science,* Vol. 615, pp. 33-56.

Coad, P. (1992) Object-Oriented Patterns. In *Communications of the ACM,* Vol. 35, pp. 152-159.

Coad, P.*, et al.* (1995) *Object Models: Strategies, Patterns, and Applications,* Prentice-Hall, Upper Saddle River, New Jersey, USA.

Cooper, J. W. (2000) *Java Design Patterns,* Addison-Wesley, Upper Saddle River, New Jersey, USA.

Coplien, J. (2000a) C++ Idioms Patterns. In *Pattern Languages of Program Design 4* (Eds, Harrison, N.*, et al.*), Addison-Wesley, Reading, Massachusetts, USA, pp. 167-197.

Coplien, J. O. (1992) *Advanced C++ Programming Styles and Idioms,* Addison-Wesley, Reading, Massachusetts.

Coplien, J. O. (1996) *Software Patterns,* SIGS Books, New York.

Coplien, J. O. (1998a) The Column Without a Name: Space, the Final Frontier. In *C++ Report,* Vol. 10, pp. 11-17.

Coplien, J. O. (1998b) The Column Without a Name: Worth a Thousand Words. In *C++ Report,* Vol. 10, pp. 51-54, 71.

Coplien, J. O. (1998c) The Geometry of C++ Objects. In *C++ Report,* Vol. 10, pp. 40-44.

Coplien, J. O. (1999) Take Me Out to the Ball Game. In *C++ Report,* Vol. 11, pp. 52-58.

Coplien, J. O. (2000b) *Multi-Paradigm Design*, PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium.

Coplien, J. O. (2003) *Promotion Ladder*, Personal correspondence to Winn, T., October 3, 2003.

Coplien, J. O. and Harrison, N. B. (2004) *Organizational Patterns of Agile Software Development,* Prentice Hall, Upper Saddle River, New Jersey, USA.

Coplien, J. O. and Schmidt, D. C. (Eds.) (1995) *Pattern Languages of Program Design,* Addison-Wesley, Reading, Massachusetts, USA.

Coplien, J. O. and Zhao, L. (2001) Symmetry Breaking in Software Patterns. In *Springer Lecture Notes in Computer Science Series (LNCS),* Vol. 2177, pp. 37-54.

Crawford, J*., et al.* (1996) Symmetry-Breaking Predicates for Search Problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning* (Eds, Aiello, L. C*., et al.*), pp. 148-159. Conference in Cambridge, MA.

Cunningham, W. (1995) The CHECKS Pattern Language of Information Integrity. In *Pattern Languages of Program Design* (Eds, Coplien, J. O. and Schmidt, D. C.), Addison-Wesley, Reading, Massachusetts, USA, pp. 146-155.

Cunningham, W. (2003) *History of Patterns*, <http://c2.com/cgi-bin/wiki?HistoryOfPatterns>, accessed June 20, 2003.

Czarnecki, K. and Eisenecker, U. W. (2000) *Generative Programming: Methods, Tools, and Applications,* Addison-Wesley, Boston, Massachusetts, USA.

Dasgupta, S. (1991) *Design Theory and Computer Science,* Cambridge University Press, Cambridge.

Dijkstra, E. W. (1985) *EWD936*, <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD09xx/EWD936.html>, accessed October 2, 2006.

Dijkstra, E. W. (1986) *EWD993*, <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD09xx/EWD993.html>, accessed October 2, 2006.

Dijkstra, E. W. (1996) *EWD1250*, <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1250.html>, accessed October 2, 2006.

Dijkstra, E. W. (2000-2001) *EWD1303*, <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1303.html>, accessed October 2, 2006.

Dooley, K. J. (1997) A Complex Adaptive Systems Model of Organizational Change. In *Nonlinear Dynamics, Psychology, and Life Sciences,* Vol. 1, pp. 69-97.

Eden, A. H.*, et al.* (1999) Department of Information Technology, Uppsala University.

Edgerton, H. E. (1957).

Eggenschwiler, T. and Gamma, E. (1992) ET++SwapsManager: Using Object Technology in the Financial Engineering Domain. In *ACM SIGPLAN Notices,* Vol. 27, pp. 166-177.

Fahle, T.*, et al.* (2001) Symmetry Breaking. In *Lecture Notes in Computer Science,* Vol. 2239, pp. 93-107.

Fincher, S. (1999) What is a Pattern Language? In *Proceedings of the Interact'99: Seventh IFIP Conference on Human-Computer Interaction*, pp. Conference in Edinburgh, UK.

Fowler, M. (1999) *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, Reading, Massachusetts, USA.

Gamma, E.*, et al.* (1993) Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Lecture Notes in Computer Science,* Vol. 707, pp. 406-431.

Gamma, E.*, et al.* (1995) *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley, Reading, Massachusetts, USA.

Gent, I. P. and Smith, B. M. (2000) Symmetry Breaking in Constraint Programming. In *Proceedings of the European Conference on Artificial Intelligence* (Ed, Horn, W.), pp. 599-603. Conference in Berlin, Germany.

Grabow, S. (1983) *Christopher Alexander: The Search for a New Paradigm in Architecture,* Oriel Press, Stocksfield, Northumberland, England.

Grenander, U. (1996) *Elements of Pattern Theory,* Johns Hopkins University Press, Baltimore, Maryland, USA.

Harandi, M. T. and Young, F. H. (1998) Software Design using Reusable Algorithm Abstractions. In *Proceedings of the IEE/BCS Conference*, pp. 93-97. Conference in Liverpool, UK.

Harris, A. (2004) *Mechanical Forces, Geometrical Shapes, and Asymmetry*, <http://www.bio.unc.edu/courses/2003spring/biol104/lecture22.html>, accessed July 30, 2004.

Harrison, N. B. and Coplien, J. O. (1996) Patterns of Productive Software Organizations. In *Bell Labs Technical Journal,* Vol. 1, pp. 138-145.

Hiroshi, Y. (2001) *A Pattern Language for Reading Books for Your Children*, <http://www.objectclub.jp/technicaldoc/pattern/readingbook>, accessed March 9, 2004.

International Organization for Standardization (2004) *Welcome to ISO Online*, <http://www.iso.ch>, accessed March 9, 2004.

Keller, W. and Coldewey, J. (1998) Accessing Relational Databases. In *Pattern Langages of Program Design 3* (Eds, Martin, R., *et al.*), Addison-Wesley, Reading, Massachusetts, USA, pp. 313-343.

Kimball, J. W. (2004) *Frog Embyology*, <http://users.rcn.com/jkimball.ma.ultranet/BiologyPages/F/FrogEmbryology.html>, accessed July 30, 2004.

Kodituwakku, S. R. and Bertok, P. (2002) Pattern Categories: A Mathematical Approach for Organizing Design Patterns. In *Proceedings of the Third Asia-Pacific Conference on Pattern Languages of Programs (KoalaPLoP 2002)* (Ed, Noble, J.), pp. 63-73. Conference in Melbourne, Australia.

Kruchten, P. (2000) *The Rational Unified Process: An Introduction,* Addison-Wesley, Boston, Massachusetts, USA.

Larman, C. (1998) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Upper Saddle River, New Jersey, USA.

Lea, D. (1994) Christopher Alexander: An Introduction for Object-Oriented Designers. In *Software Engineering Notes,* Vol. 19, pp. 39-46.

Ledermann, W. (1964) *Introduction to the Theory of Finite Groups,* Interscience, New York, New York, USA.

Lessico Publishing Group, L. (2004) *Dictionary.com*, <http://www.dictionary.com>, accessed March 9, 2004.

Limburg, K. E., *et al.* (2002) Complex Systems and Valuation. In *Ecological Economics,* Vol. 41, pp. 409-420.

Lippert, M., *et al.* (2003) Developing Complex Projects Using XP with Extensions. In *IEEE Computer,* Vol. 36, pp. 67-73.

Meszaros, G. (1995) Pattern: Half-object+Protocol (HOPP). In *Pattern Languages of Program Design* (Eds, Coplien, J. O. and Schmidt, D. C.), Addison-Wesley, Reading, Massachusetts, USA, pp. 129-132.

Meszaros, G. and Doble, J. (1998) A Pattern Language for Pattern Writing. In *Pattern Languages of Program Design 3* (Eds, Martin, R., *et al.*), Addison-Wesley, Reading, Massachusetts, USA, pp. 529-574.

Molin, P. and Ohlsson, L. (1998) The Points and Devisations Pattern Language of Fire Alarm Systems. In *Pattern Languages of Program Design 3* (Eds, Martin, R., *et al.*), Addison-Wesley, Reading, Massachusetts, USA, pp. 431-445.

Montgomery, W. (1998) *as quoted in Coplien, James O. Worth a Thousand Words, C++ Report 10(5):51-54*, Personal correspondence to Coplien, J. O.,

Naur, P. and Randell, B. (Eds.) (1969) *Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 1968,* Scientific Affairs Division, NATO, Brussels.

Noble, J. (1998a) Classifying Relationships Between Object-Oriented Design Patterns. In *Proceedings of the Australian Software Engineering Conference (ASWEC)* (Ed, Grant, D. D.), pp. 98-109. Conference in Melbourne, Victoria, Australia.

Noble, J. (1998b) Towards a Pattern Language for Object Oriented Design. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS Pacific)*, pp. 2-13. Conference in Melbourne, Victoria, Australia.

Noble, J. and Biddle, R. (2002) Patterns as Signs. In *Lecture Notes in Computer Science,* Vol. 2374, pp. 368-391.

OMG, O. M. G. (1997-2003) *Introduction to OMG's Unified Modeling Language (UML)*, <http://www.omg.org/gettingstarted/what_is_uml.htm>, accessed June 30, 2003.

Oster, G. F. and Wilson, E. O. (1978) *Caste and Ecology in the Social Insects,* Princeton University Press, Princeton, New Jersey.

Ostwald, J. (2002) *Meta-Design*, <http://webguide.cs.colorado.edu:9080/entwine/Concepts>, accessed January 13, 2004.

Pickett, J. P. (Ed.) (2000) *The American Heritage Dictionary of the English Language,* Houghton Mifflin Company, Boston.

Porter, R., *et al.* (2005) Sequences as a Basis for Pattern Language Composition. In *Science of Computer Programming Special Issue on New Software Composition Concepts,* Vol. 56, pp. 231-249.

Pree, W. (1995) *Design Patterns for Object-Oriented Software Development,* Addison-Wesley, New York.

Pressman, R. S. (1997) *Software Engineering: A Practitioner's Approach,* McGraw-HIll, New York, New York, USA.

Raskin, J. (2000) *The Humane Interface: New Directions for Designing Interactive Systems,* Addison-Wesley, Reading, Massachusetts, USA.

Reed, P. R. J. (2002) *Developing Applications with JAVA and UML,* Addison-Wesley, Boston, Massachusetts, USA.

Riehle, D. and Gross, T. (1998) Role Model Based Framework Design and Integration. In *ACM SIGPLAN Notices,* Vol. 33, pp. 117-133.

Rosen, J. (1975) *Symmetry Discovered: Concepts and Applications in Nature and Science,* Cambridge University Press, London, UK.

Rosen, J. (1995) *Symmetry in Science: an Introduction to the General Theory,* Springer-Verlag, New York, New York, USA.

Salingaros, N. A. (1995) The Laws of Architecture From a Physicist's Perspective. In *Physics Essays,* Vol. 8, pp. 638-643.

Salingaros, N. A. (2000) The Structure of Pattern Languages. In *Architectural Research Quarterly,* Vol. 4, pp. 149-161.

Savitch, W. (2004) *Absolute Java,* Pearson/Addison-Wesley, Boston, Massachusetts, USA.

Schmidt, D. C.*, et al.* (1996) Software Patterns. In *Communications of the ACM,* Vol. 39, pp. 37-39.

Senge, P. M. (1990) *The Fifth Discipline: the Art and Practice of the Learning Organization,* Doubleday, New York, New York, USA.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discpline,* Prentice Hall, Upper Saddle River, New Jersey, USA.

Stewart, I. and Golubitsky, M. (1992) *Fearful Symmetry: Is God a Geometer?,* Blackwell Publishers, Oxford, UK.

Storch, D. and Gaston, K. J. (2002) *Untangling Ecological Complexity on Different Scales of Space and Time*, Center for Theoretical Study, Charles University, CTS-02-07, July 2002.

Sullivan, K. J.*, et al.* (1996) Evaluating The Mediator Method: Prism as a Case Study. In *IEEE Transactions on Software Engineering,* Vol. 22, pp. 563-579.

Tonhouse, G. D. (1997-2002) *Nature Photography by Gary D. Tonhouse*, <http://www.reflectiveimages.com/PearlsPrairie.htm>, accessed August 1, 2006.

Vihavainen, J. (2001) *C++ Programming Techniques and Idioms*, <http://www.cs.helsinki.fi/u/vihavain/s01/cpp/idioms.html>, accessed June 2, 2003.

Viljamaa, P. (1995) The pattern business: Impressions from PLoP-94. In *Software Engineering Notes,* Vol. 20, pp. 74-78.

Walker, J. (2003) *Null Object Design Pattern*, <http://www.cs.oberlin.edu/~jwalker/nullObjPattern>, accessed December 10, 2004.

Weinand, A.*, et al.* (1988) ET++ - An Object-Oriented Application Framework in C++. In *ACM SIGPLAN Notices,* Vol. 23, pp. 47-57.

Weiss, M. (2003) Pattern-Driven Design of Agent Systems: Approach and Case Study. In *Lecture Notes in Computer Science,* Vol. 2681/2003, pp. 711-723.

Wikimedia Foundation Incorporated (2006) *Essential Complexity: from Wikipedia, the Free Encyclopedia*, <http://en.wikipedia.org/wiki/Essential_complexity>, accessed November 10, 2007.

Williams, L. and Cockburn, A. (2003) Agile Software Development: It's about Feedback and Change. In *IEEE Computer,* Vol. 36, pp. 39-43.

Winn, T. and Calder, P. (2002) Is This a Pattern? In *IEEE Software,* Vol. 19, pp. 59-66.

Winograd, T. and Flores, F. (1986) *Understanding Computers and Cognition,* Addison-Wesley, New York, New York, USA.

Wirfs-Brock, A.*, et al.* (1990) Panel: Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks. In *ACM SIGPLAN Notices,* Vol. Special Issue OOPSLA-ECOOP'90 Addendum to the Proceedings, pp. 19-24.

Wirth, N. (1974) On the Composition of Well-Structured Programs. In *Computer Surveys,* Vol. 6, pp. 247-259.

Zhao, L. and Coplien, J. O. (2002) Symmetry in Class and Type Hierarchy. In *Proceedings of the Fortieth International Conference on Technolgy of Objects, Languages and Systems (TOOLS Pacific 2002)* (Eds, Noble, J. and Potter, J.), pp. 181-190. Conference in Sydney, New South Wales, Australia.

Zhao, L. and Coplien, J. O. (2003) Understanding Symmetry in Object-Oriented Languages. In *Journal of Object Technology,* Vol. 2, pp. 123-134.

Zimmer, W. (1995) Relationships Between Design Patterns. In *Pattern Languages of Program Design* (Eds, Coplien, J. O. and Schmidt, D. C.), Addison-Wesley, Reading, Massachusetts, USA, pp. 345-364.