Towards a Pattern Language for Object Oriented Design

James Noble
Microsoft Research Institute
Macquarie University
Sydney, Australia
kjx@mri.mq.edu.au

Abstract

Since the publication of the Design Patterns book, a large number of design patterns have been identified and codified. Unfortunately, these patterns are mostly organised in an ad hoc fashion, making it hard for programmers to know which pattern to apply to any particular problem. We have organised a large number of existing object oriented design patterns into a pattern language, by analysing the patterns and the relationships between them. Organising patterns into languages has the potential to make large collections of patterns easier to understand and to use.

1: Introduction

A object oriented design pattern is a "*description of communicating objects and classes that are customised to solve a general design problem in a particular context*" [12, p.3]. Designers can incorporate patterns into their programs to address general problems in the structure of their programs' designs.

Before they can apply a pattern to solve their design problem, programmers must select an appropriate design pattern. An expert programmer may have learnt tens or hundreds of patterns, and will intuitively select the correct pattern. Novice programmers will know far fewer patterns, and will have to search pattern catalogues such as *Design Patterns* [12], *Patterns of Software Architecture* [7], or the *Pattern Languages of Program Design* series [9, 24, 14] to select a pattern. We call the problem of selecting the pattern to apply to a given design problem the *pattern selection problem*.

The first collection of patterns was made by an architect, Christopher Alexander, and these patterns described building design rather than software design [1, 2]. Alexander solved the pattern selection problem by organising his patterns into a *pattern language*. A pattern language is organised from the most general large-scale patterns to the most specific small-scale patterns, based on the relationships between the patterns. Applying the patterns in a language should assist a programmer (or architect) to generate a design, with the appropriate pattern to apply next being indicated by the organisation of the patterns in the language, that is, by the interrelationships between them [8]. In effect, the patterns in a pattern language simultaneously solve design problems and pattern selection problems.

In this paper, we describe how the patterns from the *Design Patterns* can be organised into a pattern language. Section 2 briefly reviews patterns, pattern catalogues, and pattern languages, and discusses why catalogues such as *Design Patterns* do not form pattern languages. Section 3 then outlines the organisation of the *Found Objects* pattern language which we have constructed, based upon *Design Patterns*, presenting first the large scale structure of the whole language and the small scale structure of several of the more important fragments of the language. Section 4 compares pattern catalogues and pattern languages, and Section 5 presents our conclusions.

2: Patterns, Catalogues, and Languages

A design pattern is an abstraction from a concrete recurring solution that solves a problem in a certain context [12, 7]. Typically, a design pattern has a name, a description of problems for which the pattern is applicable, an analysis of the *forces* (the important considerations and consequences of using the pattern) the pattern addresses, a sample implementation of the pattern's solution, a disussion of how this solution resolves the forces, references to known uses, and a list of patterns which are related to this pattern. To use a design pattern, a designer must first recognise a problem within their design, locate a design pattern which resolves the problem, and then design (or redesign) their program to incorporate the pattern.

Design patterns were first applied to software by Kent Beck and Ward Cunningham [5]. They were popularised by the *Design Patterns* catalogue, which described twenty-three general purpose patterns for object oriented design. Since the publication of *Design Patterns* a large number of other patterns have been identified [9, 24, 14].

2.1: Pattern Catalogues and Systems

A single design pattern generally addresses a single design problem. To provide a greater coverage of the problems faced in software development, patterns are often collected into catalogues or systems. For example, *Design Patterns* is structured as a pattern catalogue. The patterns are placed into three chapters, containing creational patterns, structural patterns, and behavioural patterns, based on the patterns' scope. Other pattern catalogues have different organisational structures. For example, *Patterns of Software Architecture* structures patterns into a system with three main categories (architecture patterns, design patterns, and programming patterns) based on the scale of the patterns. Patterns have also been catalogued based on the roles objects play in the patterns [21], patterns' internal structure [28], and the purpose of the patterns [23].

However they may be organised, pattern catalogues do not really address the pattern selection problem. First, a programmer needing to use a pattern must understand the classification scheme used by the catalogue. Second, they must search that part of the catalogue to find the pattern(s) which are applicable to their problem. Finally, although the patterns within the catalogue may point the programmer to other possible patterns which could be applied next, or could be alternatives to a particular pattern, this guidance is only at the level of the patterns, and is not part of the structure of the catalogue itself.

2.2: Pattern Languages

A pattern catalogue contains a collection of patterns which provide solutions to a collection of problems. In contrast, a *pattern language* is a collection of interrelated patterns organised into a coherent whole, which provides a detailed solution to a large-scale design problem [8, 1]. In a pattern language, the patterns are organised by the relationships between the patterns, whereas in a pattern catalogue the patterns are organised by classification schemes originating outside the patterns themselves.

The structure of a pattern language is a rooted, directed graph, generally with few cycles, where nodes represent patterns and links the relationships between patterns. The *initial pattern* at the root of the graph addresses the large-scale problem addressed by the whole language, and broadly outlines the solution the language provides. This pattern provides a partial solution to the problem, resolving some of the forces acting on the problem, but leaving some forces unresolved and exposing smaller-scale subproblems. The initial pattern is related to (it *uses* or contains) smaller-scale patterns in the language, in particular, it will use those patterns which address the subpatterns and forces exposed by the initial pattern. These patterns will in turn provide solutions, exposing further subproblems, and use smaller-scale patterns to solve them.

This organisation gives a pattern language its overall shape and is why pattern languages may provide more leverage than single patterns or pattern catalogues in addressing the pattern selection problem. Unlike a catalogue, a pattern language can be traversed by following the *uses* relationship from larger-scale to smaller-scale patterns, with each pattern both describing a solution to a subproblem, and indicating subsequent applicable patterns. In Alexander's terminology, traversing the pattern language *generates* a design [2, 1, 8]. Because of this organisation, it is more difficult to compile a pattern language than a pattern catalogue.

The progression from larger to smaller scale patterns defines the large scale structure of a pattern language, with the *uses* relationship between patterns defining the small scale structure. Larger pattern languages also have medium scale structure. Alexander's pattern language [1] is actually made up of thirty-six *pattern language fragments* — groups of between four and ten patterns which are tightly interrelated. Also, *A Pattern Language* is subtitled "*Towns · Buildings · Construction*", as the patterns (and pattern language fragments) are organised at three different scales — town planning, architecture and construction and interior decoration.

A number of pattern languages have been written for software development [9, 24, 14], but these mostly apply to particular application domains. Only a few of these "languages" contain more than ten or twelve patterns, that is, they would be better described as pattern language fragments rather than full pattern languages. To the best of our knowledge, no substantial pattern language organising object-oriented software design patterns exists.

3: Towards A Pattern Language for Object Oriented Design

We are engaged in a long-term project called "*Found Objects*" that aims to organise patterns for object-oriented design [16, 17, 18, 19]. As part of this project, we are organising a pattern language for object oriented design derived from the twenty three patterns from *Design Patterns*, and including a number of other patterns drawn from the general patterns literature [7, 4, 9, 24, 14]. The current version of the pattern language, called *Found Objects*, contains over 90 patterns. Although this is approximately three times as many patterns as the *Design Patterns* catalogue, the language is not three times as complex, because the patterns in the language are smaller than those in *Design Patterns*. As part of organising the patterns into a language, we have subdivided the larger patterns, so that each pattern focuses on addressing one particular problem, and to explicate latent information in the pattern descriptions.

Because of the size of the pattern language, space does not permit us to present it in full detail here. To give the flavour of the language, we begin by presenting an overview of the language's large scale structure. We then present the structure of some of the more important fragments of the language in detail, to illustrate how patterns can be organised into pattern langauge fragments and then pattern languages. Also for space reasons, and since we have built the language primarily by organising well-known patterns from the pattern literature, we do not describe individual patterns in detail.

3.1: Large Scale Structure

The *Found Objects* pattern language is constructed out of a number of *pattern language fragments*. Each pattern language fragment contains a number of related patterns, and the relationships between the patterns in different language fragments determines the structure of the language as a whole. Figure 1 illustrates the structure of the fragments in the language, and shows how the fragments can be loosely organised into three main categories — architectural patterns, design patterns, and programming idioms — following *Patterns of Software Architecture* [7]. The links between the language fragments in Figure 1 represent the major dependencies between the patterns in the fragments — patterns in the higher level fragments use the patterns in the lower-level fragments. Most fragments contain between five and ten patterns, although some (in particular, Interpreter) contain only a single pattern. Figure 2 gives an overview of the patterns in each fragment.

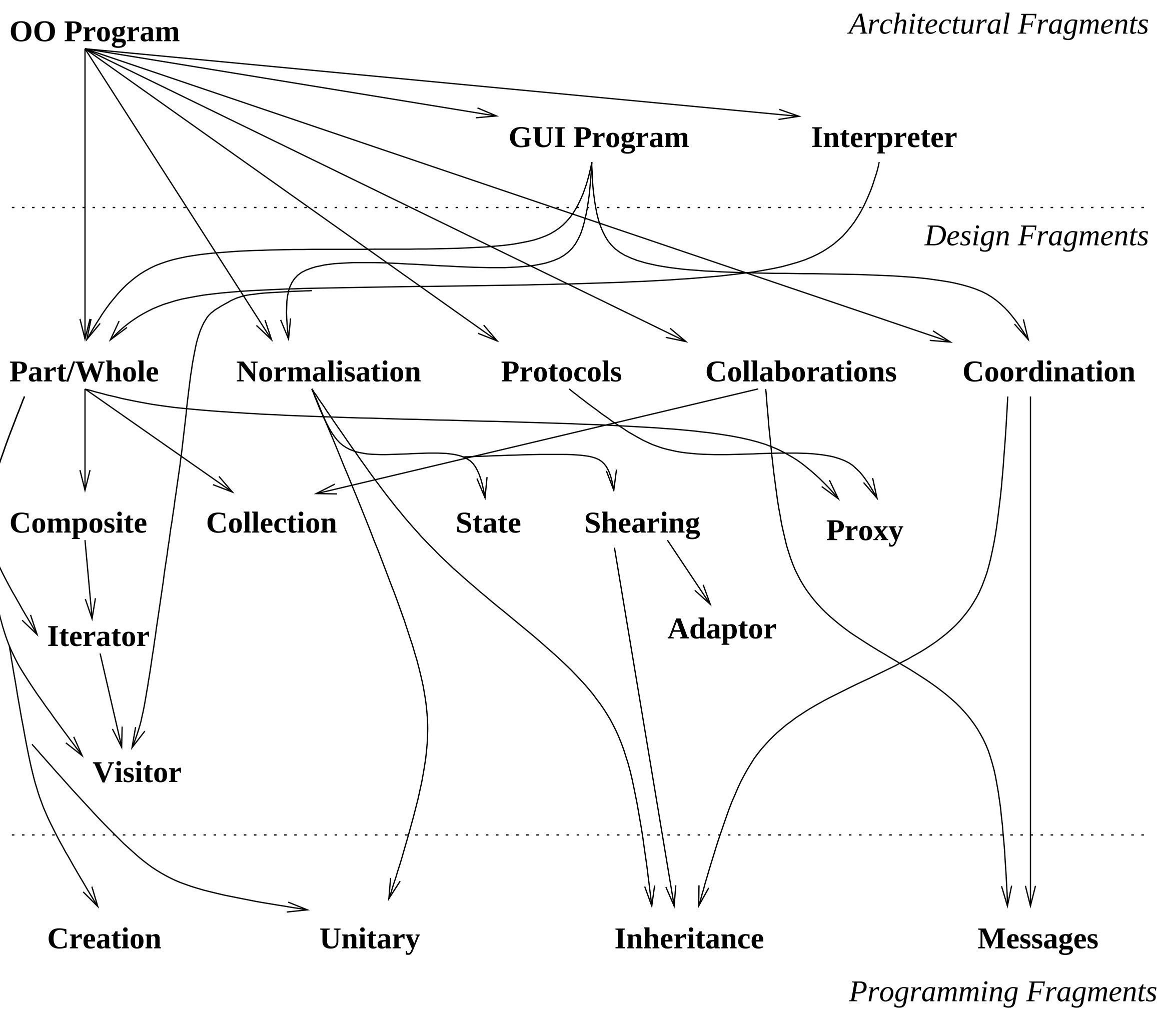

Figure 1. The Structure of the Pattern Language

The language has three architectural pattern fragments. The most important fragment is the OO Program fragment. This is the initial fragment, and it contains the initial pattern which is also called OO Program. The other architectural fragments describe architectural composite patterns

Fragment	Patterns
Architectural Fragments	
OO Program	OO program, Objects, Responsibilities, Collaborations [25]
GUI Program	MVC, View Handler, Command Processor [7]
Interpreter	Interpreter [12]
Design Fragments	
Part/Whole	Aggregation, Composition, Sharing [7]
Normalisation	Type Object [13], Method Object [4], State [10]
Protocols	Patterns on protocol design [17]
Collaborations	Patterns on relationship design [18]
Coordination	Chain of responsibility, Observer, Mediator [12]
Collection	Patterns on collections [4, Chapter 5]
Shearing	Facade, Bridge, Adaptor, Decorator, Strategy, Extension [11]
Programming Fragments	
Creation	Factory method, Abstract factory, Prototype, Builder [12]
Unitary	Singleton, Memento, Flyweight [12], What If [3], Null Object [27]
Inheritance	Abstract class [26], Template method, Hook method [12]
Messages	Delegation, OO Recursion [3], Double Dispatch [12]

Figure 2. The Major Fragments of the Language

which define large parts of a program's architecture — the GUI Program fragment contains high-level patterns for building user interfaces based on the Model-View-Controller pattern, and the Interpreter fragment contains only the Interpreter pattern, the sole larger-scale pattern from *Design Patterns*.

The majority of the patterns in the language address problems in OO design. This is unsurprising, since we are developing the language to organise the *Design Patterns*. The design section begins with five major fragments (Part/Whole, Normalisation, Protocols, Collaborations, and Coordination) containing patterns about designing objects' structures, interfaces, and relationships. The Part/Whole fragment is based upon the Part/Whole pattern [7], and leads to separate fragments which describe the Composite, Iterator, and Visitor patterns and their commonly occurring variants. The Normalisation fragment contains patterns about decomposing objects into subobjects, such as State, Type Object [13], and Method Object [4]. The Protocol and Collaboration fragments contain patterns about objects' interfaces and relationships [17, 18], the Coordination fragment contains patterns such as Mediator and Observer which coordinate or distribute control over multiple objects in programs, and the Shearing fragment contains patterns such as Strategy, Facade, and Adaptor, which help programs handle multiple rates of change within their structure.

The pattern language also includes a number of other OO design fragments named after a particular pattern. These fragments are related to the larger design patterns after which they are named, and typically contain that pattern, plus a number of smaller-scale patterns which describe how that design pattern can be implemented or alternative ways it can be used. For example, the Composite, Iterator, Visitor, and Adaptor fragments include several variants of each pattern, and the Collections fragment contains patterns which describe how collection classes can be used [4, Chapter 5].

Finally, the programming fragment contains lower level patterns which are used by many other patterns in the language, including patterns which rest solely upon inheritance, the creational patterns, and patterns which relate to unitary, self contained objects. The most interesting fragment here is the Creation fragment, which organises the Creational patterns from *Design Patterns*, in addition to other creational patterns like Product Trader [22].

3.2: Small Scale Structure

The small scale structure of the pattern language is determined by the specific patterns organised into each fragment of the pattern language, and the relationships between the patterns in each fragment. We consider three relationships between patterns — one pattern *uses* another pattern, patterns can *conflict* in providing differing solutions to common problems, and one pattern can *refine*, or be a specialisation of, another pattern [19].

Uses One pattern *uses* another pattern when the second pattern solves a sub-problem raised by the application of the first pattern. For example, the Abstract Factory pattern *uses* the Factory Method pattern because abstract factories are often implemented using factory methods. The *uses* relationship is the most important and most common relationship between the patterns. The *uses* relationship guides the programmer through the language, indicating which patterns may be applicable at any stage. The *uses* relationship is the only explicit relationship between patterns in *A Pattern Language* [1] (where it is called *containment*), and most software pattern forms also explicitly record this relationship — typically in a section titled *Related Patterns* [12] or *See Also* [7]. Some pattern forms, including Alexander's, also record the inverse *used-by* relationship to give the context of more general patterns within which a particular pattern is likely to be instantiated.

Conflicts Two or more patterns can *conflict*, that is, provide mutually exclusive solutions to similar problems. For example, the Decorator pattern conflicts with the Strategy pattern in that both patterns can (and have been) be used to add graphical borders or icons to window objects in window systems [12, p.180]. Most pattern forms do not provide an explicit section to record this relationship, but it is often expressed in the related pattern section along with the *uses* relationship or it may be discussed elsewhere in the pattern form.

Refines One pattern can *refine* another pattern, that is to say, one pattern can be a specialisation of another pattern. For example the **Factory Method** pattern *refines* the Hook Method pattern, and in *A Pattern Language* the Sequence of Sitting Spaces pattern refines the Intimacy Gradient pattern [1]. A specific pattern refines an abstract pattern if the specific pattern's full description is a direct extension of the abstract pattern. That is, the specific pattern must deal with a specialisation of the problem the general pattern addresses, must have a similar (but more specialised) solution structure, and must address the same forces as the more general pattern, but may also address additional forces. To make an analogy with object oriented programming, the *uses* relationship is similar to composition, while the *refines* relationship is similar to inheritance.

3.3: The Architectural Fragments

The pattern language proper begins with the initial pattern from the initial fragment of the pattern language (see Figure 3). This is the OO Program pattern, which describes the single largest artifact produced by the language, by analogy with the Independent Regions (1) pattern from *A Pattern Language*. The OO Program pattern leads to several conceptual patterns which describe how programs are built from objects, their collaborations, and their relationships, and which in turn lead to more specific patterns. The OO Program pattern also leads to the composite patterns Model-View-Controller and Interpreter (in the GUI Program fragment and the Interpreter fragment respectively) to generate the overall structure of the program.

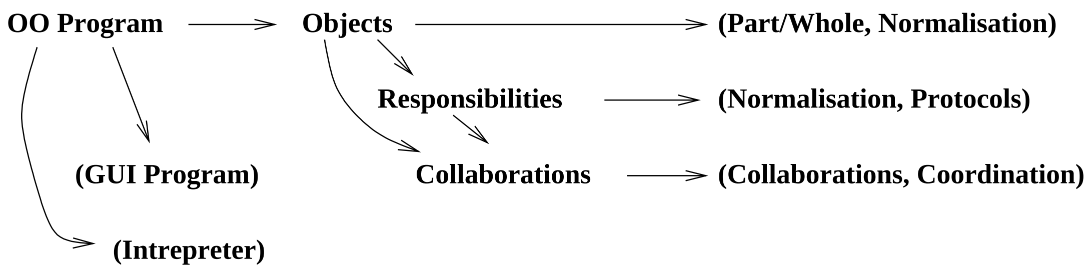
OO Program
Objects
(Part/Whole, Normalisation)
Responsibilities
(Normalisation, Protocols)
(GUI Program)
Collaborations
(Collaborations, Coordination)
(Intrepreter)

Figure 3. The OO Program fragment

Although the OO Program fragment is the capstone of the pattern language, it was one of the last parts of the language we completed, and it was the only fragment where we had to compile all the patterns specifically for the language. Once the other patterns were organised, the language needed an initial fragment as a starting point for reading or working through the patterns, so we introduced the OO Program pattern to fill this need, and the Objects, Responsibilities, and Collaborations patterns to fill it out. These patterns describe the basics of *Responsibility Driven Design* [25]. Together, these patterns provide an object oriented context in which the rest of the language can operate, and lead the reader into the more detailed design patterns.

This section also includes some composite architectural patterns. The GUI Program fragment (see Figure 4) is based around the Model-View-Controller composite pattern [7, 20], and includes the Command Processor and View Handler patterns [7]. The Model-View-Controller pattern also uses a number of smaller-scale patterns from other fragments in the language — these are shown parenthesised in the figure.

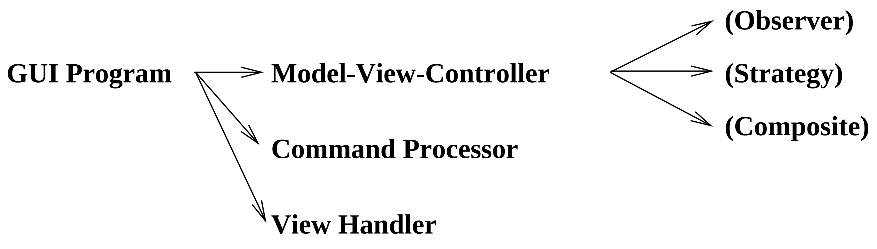

Figure 4. The GUI Program Fragment

The last architectural pattern langauge fragment contains only one pattern, Interpreter. We have placed this pattern into the architectural level of the language because it is at a higher level than the other patterns in *Design Patterns* — in particular, it can be described as a composite pattern which uses the Composite and Visitor patterns.

3.4: The Design Fragments

The Part/Whole fragment of the pattern language, illustrated in Figure 5, is the first fragment of the design patterns, and describes how larger objects can be composed from smaller parts. This fragment is based around the Part/Whole pattern from *Patterns of Software Architecture*, and is a complex fragment, because the patterns it contains have complex interrelationships. The main Part/Whole pattern is refined by three other patterns. The Assembly pattern [7] describes how aggregate objects can be assembled from smaller objects. The Collection pattern describes how collection (or container) objects can be used to hold groups of objects, and it refers the reader to a pattern language fragment describing a particular Collection library — Beck [4, Chapter 5] provides a good set for Smalltalk. The OO Trees pattern [3] describes how trees of objects can be assembled recursively using the very common Composite pattern, and also other patterns like Decorator and Visitor. Finally, the Sharing pattern [3, 7] describes how one Whole may share its Parts with other Wholes, and leads to its common specialisation, Flyweight, which is part of the Unitary programming fragment.

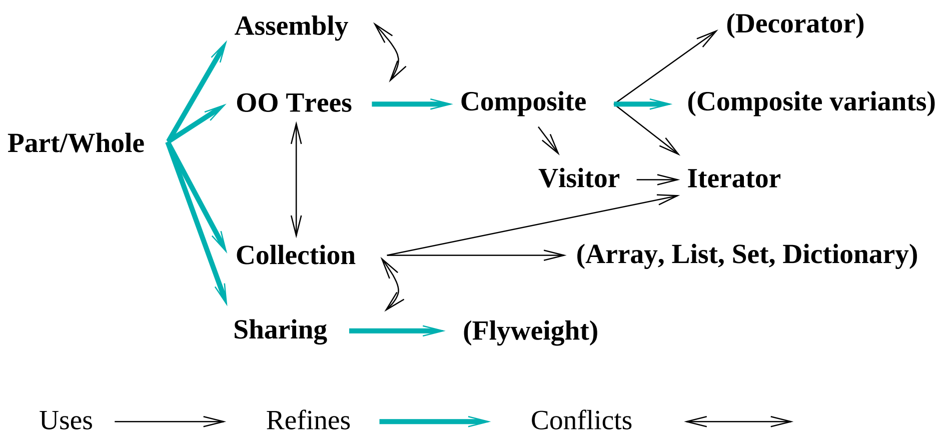

Figure 5. The Part/Whole fragment

The Part/Whole fragment illustrates how abstract patterns (such as the Part/Whole pattern) can be incorporated into a larger pattern language. The abstract Part/Whole pattern *refines* the more specific patterns it generalises, and proceeds these in the language. The Assembly, OO Trees, and Collection patterns *conflict* with each other, because, in refining Part/Whole, each provides a different solution to the general problem of decomposing an object.

3.4.1: Shearing Fragment

The Shearing Fragment organises a number of the *Design Patterns*, plus a number of patterns identified more recently. All the patterns in this fragment provide ways help programs remain flexible when different parts of their structure much change at different rates — the fragment takes its name from the *Shearing Layers* identified in buildings in *How Buildings Learn* [6]. Although other patterns also have this effect, the Shearing patterns address this problem most directly.

The fragment begins with an abstract pattern, also called Shearing, which identifies the general problem, and is refined by two conflicting patterns (Skin and Guts) which capture the dynamics of the two main solutions — “*Changing the skin of an object versus changing its guts*” [12, p. 179], that is, changing an objects interface versus changing its implementation. These two patterns are refined by more detailed patterns which provide concrete solutions in particular contexts, including the Bridge pattern, which allows both Skin and Guts to vary independently.

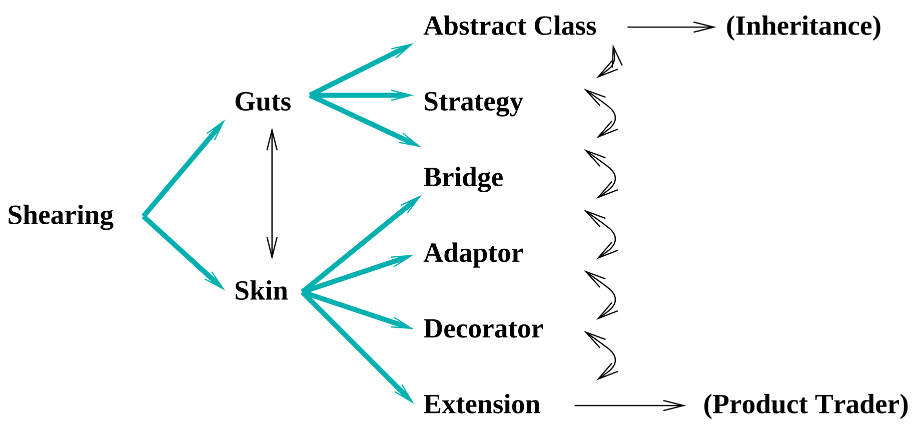

Figure 6. The Shearing Fragment

3.4.2: State Fragment

The State fragment incorporates Dyson and Anderson's *State Patterns* pattern language fragment [10] directly into our larger pattern language (see Figure 7). In this fragment, the State pattern captures the core of the State pattern from *Design Patterns*, and the other patterns act as *subpatterns* of State, describing how to apply it in more detail. In particular, the State Member and Exposed State patterns describe how to design the subsidiary state objects, the Owner Driven Transitions and State Driven Transitions patterns describe two alternative design for managing transitions between states, and the Pure State pattern describes how and when state objects can be shared. Because it is quite self-contained, this fragment can be directly incorporated into our pattern language — the patterns and their relationships are taken directly from the original description [10].

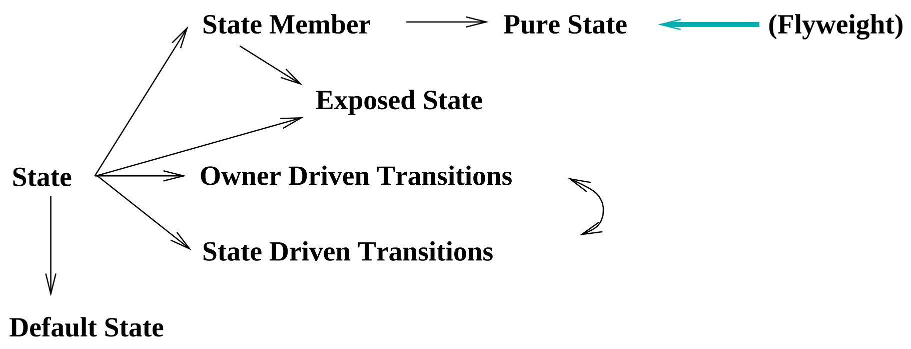

Figure 7. The State Fragment

The state fragment illustrates two points about building pattern languages. First, subpatterns can be incorporated simply by organising the language so that the that the main pattern *uses* all the top-level subpatterns. Second, well-conceived pattern language fragments can sometimes be incorporated wholesale into larger pattern languages.

3.5: The Programming Fragments

The Creation fragment is the most interesting Programming fragment, so it is the only one we present here (see Figure 8). This fragment incorporates all the *Design Patterns* creational patterns, the Product Trader pattern [22], and two abstract patterns — Natural Creation and Direct Creation — which are introduced to structure the fragment [15]. Natural Creation address the design problem of how should objects be created, and Direct Creation refines patterns describing the two basic mechanisms for creating objects provided by programming languages — instantiating a class or cloning a prototype.

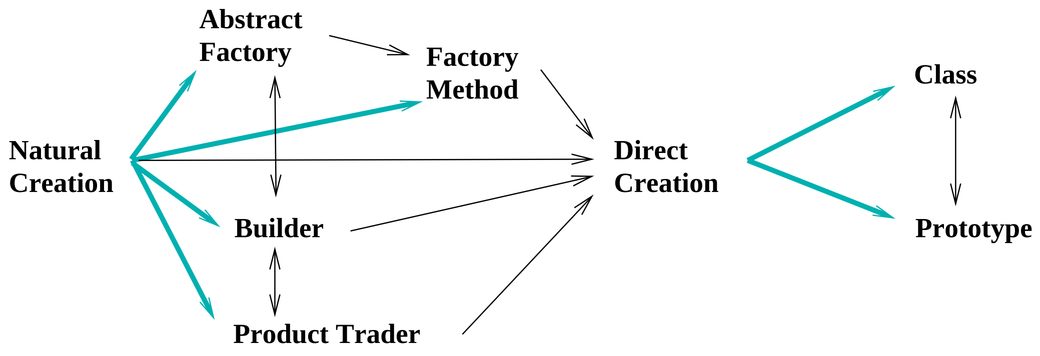

Figure 8. The Creation Fragment

4: Discussion

Design Patterns contains an analysis of why pattern catalogues are *not* pattern languages:

1. *People have been making buildings for thousands of years, and there are many classic examples to draw upon. We have been making software systems for a relatively short time, and few are considered classics.*
2. *Alexander gives an order in which his patterns should be used; we have not.*
3. *Alexander's patterns emphasise the problems they address, whereas design patterns describe the solutions in more detail.*
4. *Alexander claims his patterns will generate complete buildings. We do not claim that our patterns will generate complete programs.*

Design Patterns [12, p. 356], Gamma, Helm, Johnson, Vlissides.

In order to organise the *Design Patterns* into a language we must address these four points. We have not addressed the first point directly — there are still very few programs which are considered classics, and we have not tried to write or unearth any. In spite of this, the *Design Patterns* do seem to capture many of the important features of the design of those extant object oriented programs which are considered classics, and the patterns are becoming widely recognised as good software engineering practice.

The second point is the most important consideration for organising patterns into a language. *Design Patterns* is a pattern catalogue, so the patterns are organised into three chapters based on the pattern's scope, and within each section the sequence is alphabetical — essentially ad-hoc. In our pattern language, we have explicitly provided an order for the patterns, based on the relationships between the patterns, and the scale at which each pattern applies, to guide the programmer through the patterns.

The third point is also quite important, because although the bulk of the pattern descriptions we have drawn upon do concentrate upon the proposed solutions, all patterns include a description of the problem they solve — although in the *Design Patterns* form, it is split between the Intent, Motivation, and Applicability sections. In constructing our pattern language, we have analysed the patterns to identify the problems that each pattern solves, and where necessary decomposed monolithic patterns to highlight the problems the patterns address.

Finally, the fourth point is important, although less so than the second point. In particular, *Design Patterns* contains no larger-scale patterns to act as starting points for a pattern language, and there is certainly no initial pattern. The patterns also stop short of capturing lower-level knowledge about object oriented programming.

For organising a pattern language, the lack of higher-level patterns is more important, since they group the patterns into the whole language, and so we have introduced a number of large scale patterns to start the language. The pattern language we have constructed is an initial attempt, and is intentionally partial and open to extension. Although a complete pattern language in the sense that a path can be traced through the sequence of patterns from the initial architectural pattern to the lowest level programming pattern, the language will requires many more patterns to be able to generate designs for whole programs.

The pattern language is also incomplete in another sense, in that the we have described only the organisation or skeleton of a pattern language. This is because the language is made up mostly from existing patterns from the literature, so the body of the patterns are simply references into that literature. The complete language would be much more convenient to use if it included the full text of every pattern, but this would be a major undertaking. It is also not clear what kind of pattern form is best suited to this style of pattern language.

5: Conclusion

In this paper, we have described how the design patterns from *Design Patterns* can be organised into a pattern language, along with other patterns from the literature. We have described the large-scale structure of the resulting *Found Objects* pattern language, and outlined the contents and relationships within some of the main fragments in the language.

Practitioners and researchers need to experiment with the resulting pattern language, to evaluate the benefits and liabilities of presenting patterns using a pattern language vis-a-vis a pattern catalogue or pattern system. At this time, it is not clear whether pattern catalogues or pattern languages will prove to be the better approach for organising a practical handbook for software engineering. Organising the *Design Patterns* into a pattern language demonstrates that at least some kind of pattern language can be constructed for general, domain-independent software design patterns, and is an important step enabling more detailed comparisons to be carried out.

Acknowledgements

Thanks to Geoffrey Outhred, Ralph Johnson, and the anonymous referees, who kindly commented on drafts. This work was supported by Microsoft Australia Pty Ltd.

References

[1] Christopher Alexander. *A Pattern Language*. Oxford University Press, 1977.

[2] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison-Wesley, 1988.

[4] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[5] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical report, Tektronix, Inc., 1987. Presented at the OOPSLA-87 Workshop on Specification and Design for Object-Oriented Programming.

[6] Steward Brand. *How Buildings Learn*. Penguin Books, 1994.

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[8] James O. Coplien. *Software Patterns*. SIGS Management Briefings. SIGS Press, 1996.

[9] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1996.

[10] Paul Dyson and Bruce Anderson. State objects. In Martin et al. [14].

[11] Erich Gamma. Extension object. In Martin et al. [14].

[12] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[13] Ralph E. Johnson and Bobby Woolf. Type object. In Martin et al. [14].

[14] Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1998.

[15] Gerard Meszaros and Jim Doble. A pattern language for pattern writing. In Martin et al. [14].

[16] James Noble. Found objects. In *EuroPLOP Proceedings*, 1996.

[17] James Noble. Arguments and results. In *PLOP Proceedings*, 1997.

[18] James Noble. Basic relationship patterns. In *EuroPLOP Proceedings*, 1997.

[19] James Noble. Classifying relationships between object-oriented design patterns. In *Australian Software Engineering Conference (ASWEC)*, 1998.

[20] Dirk Riehle. Composite design patterns. In *ECOOP Proceedings*, 1997.

[21] Dirk Riehle. A role based design pattern catalog of atomic and composite patterns structured by pattern purpose. Technical Report 97-1-1, UbiLabs, 1997.

[22] Dirk Riehle. Product trader. In Martin et al. [14].

[23] Walter F. Tichy. A catalogue of general-purpose software design patterns. In *TOOLS USA 1997*, 1997.

[24] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[25] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[26] Bobby Woolf. The abstract class pattern. In *PLOP Proceedings*, 1997.

[27] Bobby Woolf. Null object. In Martin et al. [14].

[28] Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.