Organising Patterns into Languages
Towards a Pattern Language for Object Oriented Design

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

April 14, 1998

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 design pattern is a "*description of communicating objects and classes that are customised to solve a general design problem in a particular context*" [19, 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 for a given problem. Novice programmers will know far fewer patterns, and will have to search pattern catalogues such as *Design Patterns* [19], *Patterns of Software Architecture* [8], or the *Pattern Languages of Program Design* series [11, 35, 26] to select a pattern.

To address this problem, Christopher Alexander organised his architecture and construction patterns into a *pattern language*. A pattern language finesses the *pattern selection problem*, because 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. By applying the patterns in a language, a programmer should be able to generate a design from scratch, with the appropriate pattern to apply next being determined by the organisation of the patterns in the language, that is, by the interrelationships between them [1, 2]. 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 architecture of the *Found Objects* pattern language which we have constructed, based upon *Design Patterns*, and Section 4 briefly describes the process we used to organise the patterns into the language. Finally, Section 5 discusses our work, and Section 6 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 [19, 8]. 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, 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 [11, 35, 26].

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 by 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 [31], patterns' internal structure [39], and the purpose of the patterns [34].

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 [10, 22, 1]. In a pattern language, the patterns are organised by the relationships between the patterns, whereas in an 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*) 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. Unlike a catalogue, a pattern language can be traversed by following the *uses* relationship from larger-scale to smaller-scale patterns, which 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, 10]. Because of this structure, it is more difficult to organise patterns into a language than into a 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, but these mostly apply to particular application domains, such as user interfaces [12], connecting relational databases to object oriented systems [21, 7], or software process [9, 13]. 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 general-purpose software design patterns exists.

3 A Pattern Language for Object Oriented Design

We have organised 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 [8, 4, 11, 35, 26]. The resulting pattern language, *Found Objects* [30], contains over 90 patterns, so space does not permit us to present the language in full detail here. Although our language includes 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.

In this section, we begin by presenting the large scale structure of the pattern language. We then describe the language's small scale structure, and present some of the more important architectural and design fragments making up the language.

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 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* [8]. 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 3.1 gives an overview of the patterns in each fragment.

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 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 have developed 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 [8], 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 [20], and Method Object [4]. The Protocol and Collaboration fragments contain patterns about objects' interfaces and relationships [28, 29], 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 or class. 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

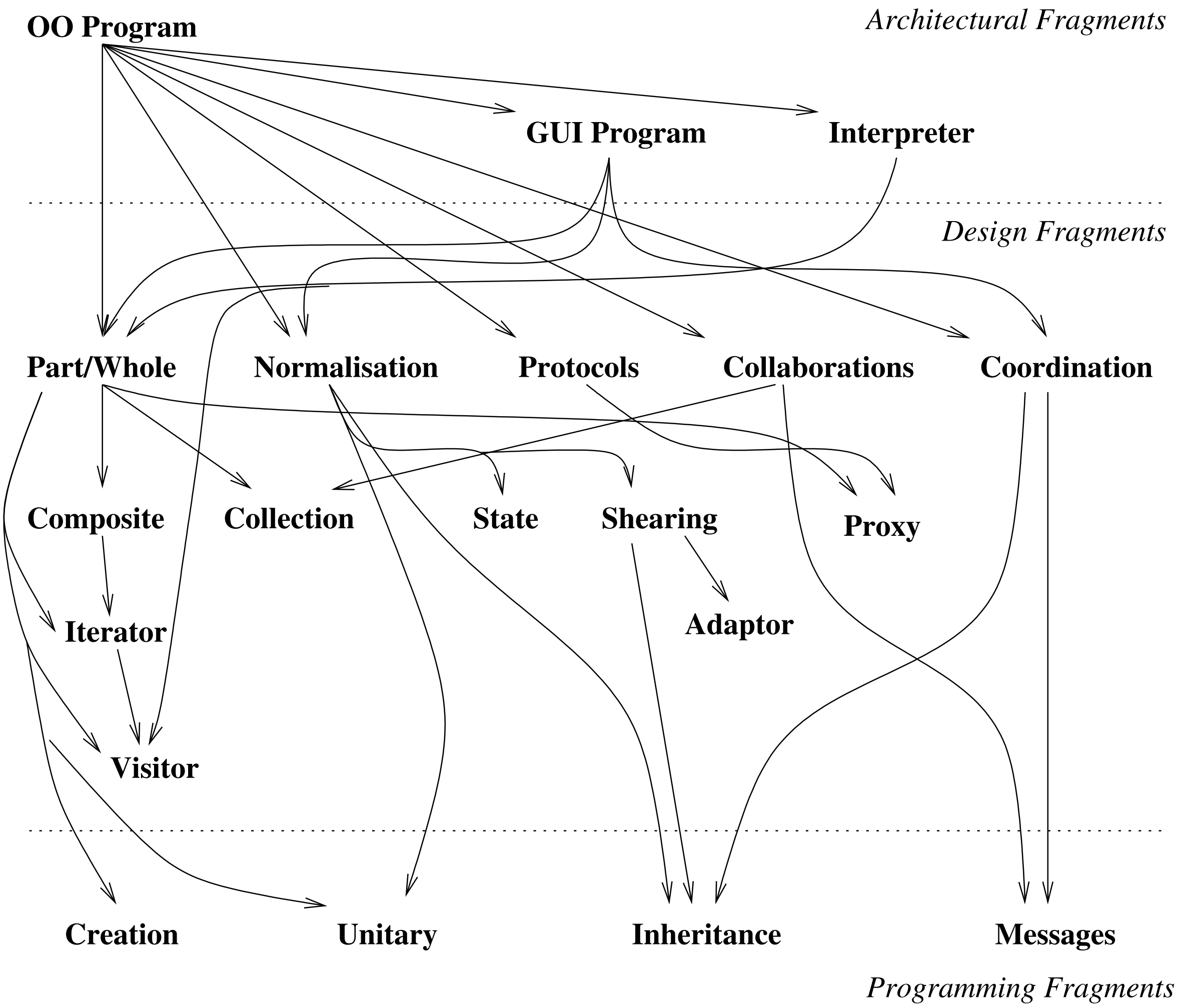

Figure 1: The Structure of the Pattern Language

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 probably the Creation fragment, which organises the Creational patterns from *Design Patterns*, in addition to other creational patterns like Product Trader [33].

Fragment	Patterns
Architectural Fragments	
OO Program	OO program, Objects, Responsibilities, Collaborations [36]
GUI Program	MVC, View Handler, Command Processor [8]
Interpreter	Interpreter [19]
Design Fragments	
Part/Whole	Aggregation, Composition, Sharing [8]
Normalisation	Type Object [20], Method Object [4], State [14]
Protocols	Patterns on protocol design [28]
Collaborations	Patterns on relationship design [29]
Coordination	Chain of responsibility, Observer, Mediator [19]
Collection	Patterns on collections [4, Chapter 5]
Shearing	Facade, Bridge, Adaptor, Decorator, Strategy, Extension [18]
Programming Fragments	
Creation	Factory method, Abstract factory, Prototype, Builder [19]
Unitary	Singleton, Memento, Flyweight [19], What If [3], Null Object [38]
Inheritance	Abstract class [37], Template method, Hook method [19]
Messages	Delegation, OO Recursion [3], Double Dispatch [19]

Figure 2: The Major Fragments of the Language

3.2 Small Scale Structure

The small scale structure of the pattern language is determined by the relationships between the patterns, and the kinds of patterns in the language.

3.2.1 Relationships between Patterns

We have organised the language using three relationships between patterns. The dominant relationship is whether one pattern *uses* another pattern, but patterns can also *conflict* in providing differing solutions to common problems, and one pattern can *refine*, or be a specialisation of, another pattern [27].

Uses 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], and most software pattern forms also explicitly record this relationship — typically in a section titled *Related Patterns* [19] or *See Also* [8]. 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 [19, 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 is a specialisation of another pattern. For example, in our pattern language **Factory Method** refines **Hook Method**, and in *A Pattern Language* the **Sequence of Sitting Spaces** pattern refines the **Intimacy Gradient** pattern [1]. A specific pattern refines a more abstract pattern if the specific pattern's full description is a direct extension of the more general 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.

3.2.2 Kinds of Patterns

We also analysed the kinds of patterns we wished to incorporate into the language. Since the publication of *Design Patterns*, many other kinds of patterns have been described — process patterns [9], analysis patterns [17], subpatterns [14], composite patterns [32], variant patterns [8], self-applicative pattern tilings [23], and abstract patterns [3]. Some of these are patterns about particular domains — in particular, analysis patterns describe analysis, and process patterns describe organisational structures. The other kinds of patterns are domain independent, so they can describe the design of object oriented programs and so need to be incorporated into the pattern language.

Composite Patterns Riehle recently defined composite patterns as "*… any design pattern which is best described as the composition of further patterns*" [32]. These *Composite Patterns* are distinct from the **Composite** pattern in *Design Patterns* — where *the* Composite pattern composes objects, *a* composite pattern composes other patterns. For example, the Model-View-Composer pattern is composed from the Composite, Strategy, and Observer patterns. A composite pattern *uses* the patterns from which it is composed.

Abstract Patterns An *abstract pattern* is a generalisation of one or more other patterns in a pattern language. The *Design Patterns Smalltalk Companion* introduces a number of abstract patterns, such as Sharable which generalises Flyweight [3, p. 197], and Recursive Delegation which generalises Chain of Responsibility [3, p. 231]. An abstract pattern *refines* the patterns it generalises.

Variant Patterns A pattern is different every time it is used, because it is instantiated to suit the particular problem it solves. Some kinds of problems occur more regularly than others, so some ways of instantiating patterns are more common than others. These common patterns of instantiation are called *variant* patterns [8]. To organise variant patterns into the pattern language, we treat each major variant as a separate pattern, which typically *refines* the main pattern and *conflicts* with mutually exclusive variants. This decomposition is important, because it helps ensure that each pattern in the language is providing one particular solution to one particular problem. In particular, we are careful to decompose patterns which provide a number of variant solutions to similar problems into separate *solution variants*, and patterns which provide similar solutions to a number of different problems into *problem variants*.

Tiling Variants Some patterns can be applied repeatedly to solve a single problem. Lorenz has identified some particular examples of this as *Pattern Tilings* [23]. We treat repeated applications of design patterns as additional solution variants, that is, a tiling variant *refines* and *uses* the main pattern.

Subpatterns Patterns and pattern language fragments have been written to describe how other design patterns can be implemented. We call these patterns *subpatterns*. For example, Dyson and Anderson have written a small pattern language of subpatterns which describe in more detail how to apply the State pattern [14]. A larger scale pattern *uses* the subpatterns which describe how it is implemented.

3.3 The Architectural Fragments

The pattern language proper begins with the initial pattern from the initial fragment (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 more basic 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.

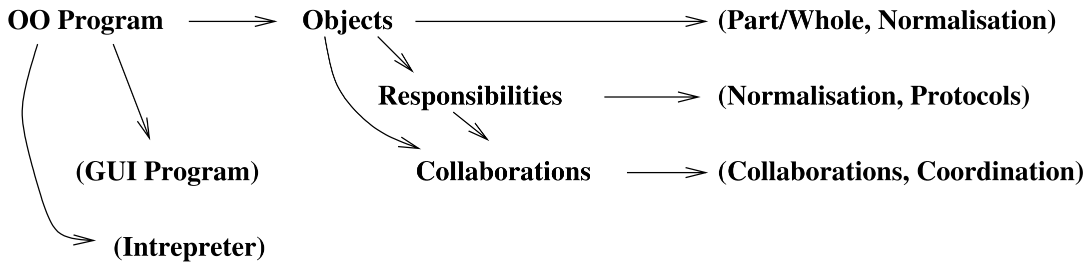

Figure 3: The OO Program fragment

Although the initial 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 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* [36]. 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 is based around the Model-View-Controller composite pattern [8, 32], and includes the Command Processor and View Handler patterns [8]. 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.

The interpreter 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

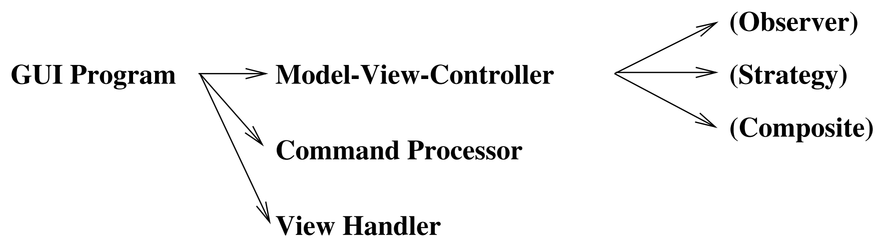

Figure 4: The GUI Program Fragment

Visitor patterns.

These fragments provide examples of how composite patterns can be incorporated into a pattern language. A composite pattern *uses* the smaller-scale patterns of which it is composed, and should precede these in the sequence of the language. A composite pattern is typically a much larger-scale pattern than the patterns it uses, so the patterns will often be in different fragments.

3.4 The Design Fragments

The Part/Whole fragment, 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 [8] 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 pattern refers the reader to a 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 [8, 3] 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.

The Part/Whole fragment illustrates how abstract patterns can be incorporated into a larger pattern language. In particular, Part/Whole is an abstract pattern, so it *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.

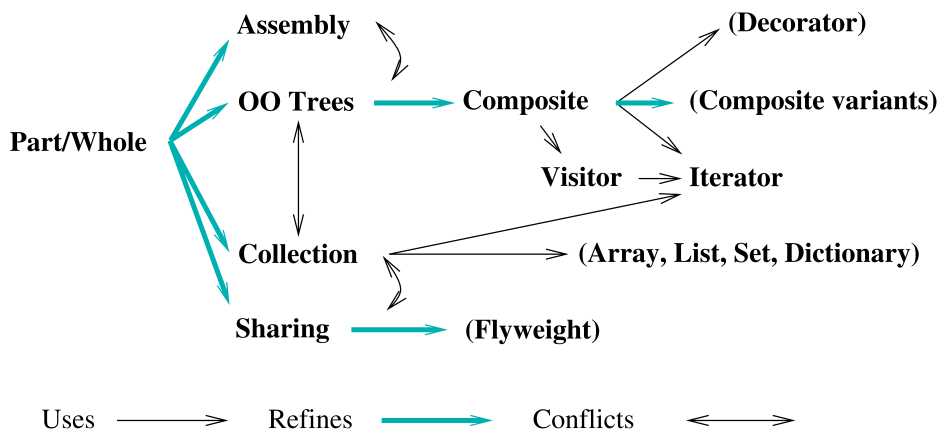

Figure 5: The Part/Whole fragment

3.4.1 Shearing Fragment

The Shearing Fragment organises a number of *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*" [19, p. 179], that is, changing an objects interface versus changing its implementation. These two patters 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.

3.4.2 Adaptor Fragment

The Shearing fragment uses the Adaptor pattern which itself has a number of solution variants — two major variants, Class Adaptor and Object Adaptor [19, p. 141], and two minor variants, Pluggable Adaptor and Two-way Adaptor [19, p. 142-143]. Figure 7 shows how the Adaptor pattern and its variants are incorporated into the pattern language. The main Adaptor pattern introduces a common problem — adapting the interface of an object — and the four solution variants are linked to it by the *refines* relationship, because they provide more specific solutions to that general problem. Class and Object Adaptor are conflicting patterns because they offer mutually exclusive solutions to any

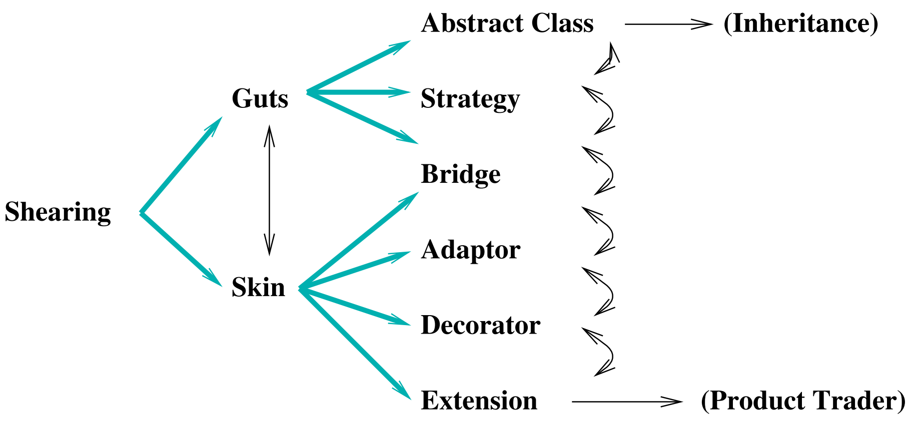

Figure 6: The Shearing Fragment

given adaption problem. Two-way Adaptor also *uses* the Class Adaptor pattern — this is discussed in section 3.4.4 below.

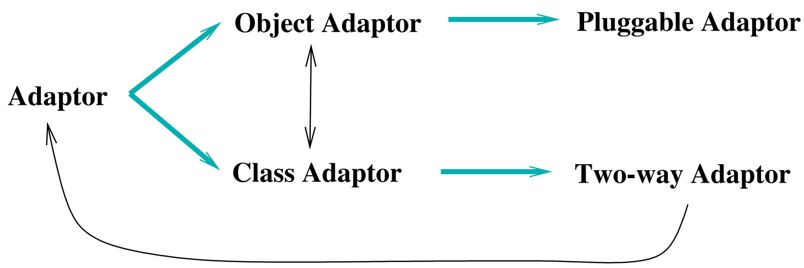

Figure 7: The Adaptor Fragment

3.4.3 Proxy Fragment

The **Proxy** pattern is described in *Design Patterns* and *Patterns of Software Architecture*, and each description introduces a number of major variants — four in *Design Patterns*, and these four plus another three in *Patterns of Software Architecture*. Basically, Proxy describes a solution — replace an object with a surrogate object — but does not describe any single problem this solution resolves. Rather, the many variants of the Proxy pattern each describe a different problem to which Proxy provides solution. *Patterns of Software Architecture* makes the problem variations explicit in its introduction to the Proxy Pattern — "*Introducing such a placeholder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorised access*" [8, p. 263].

Figure 8 shows the overall structure of the Proxy fragment. We have decomposed the monolithic

Proxy pattern so that each variant problem is captured as a separate pattern (on the left of the figure), and the main pattern then introduces the common solution. This decomposition is particularly important as it helps ensure the patterns in the resulting language focus on problems at least as much as solutions.

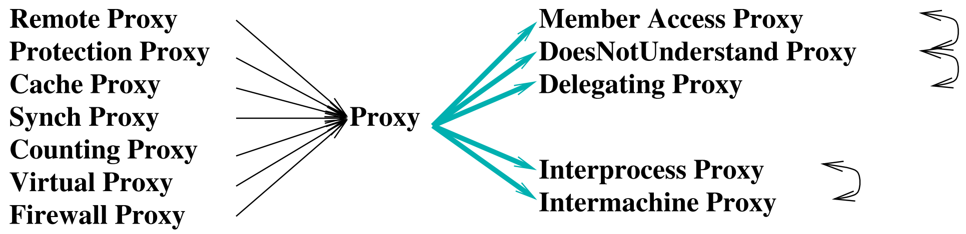

Figure 8: The Proxy Fragment

In the figure, each problem variant *uses* the basic pattern — we do not record a *conflicts* relationship between the different problem variants, or a *refines* relationship between the variant and main patterns, because these patterns all address *different* problems.

The full Proxy Fragment shown in Figure 8 is more complex than we have described here, because it also includes a number of *solution* variants (on the right) which *refine* the base Proxy pattern, in the same way the Adaptor solution variants *refine* adaptor.

3.4.4 Composite Fragment

Some patterns can be applied repeatedly to solve a single problem. Lorenz has identified some particular examples of this as *Pattern Tilings* [23]. For example, the two-way adaptor variant described in Section 3.4.2 above can be seen as a tiling of the adaptor pattern, because the Class Adaptor pattern is applied twice to the same Target and Adaptee interfaces [19, p. 143].

We treat repeated applications of design patterns as additional solution variants, that is, a tiling variant *refines* the main pattern, however a tiling variant also *uses* the main pattern. With this approach, repeated application does not need to be treated as a "*fundamental reflexive relationship*" within the pattern language [23], rather, a tiling pattern is simply a pattern which uses itself.

Recording repeated applications as tiling variants has the advantage that complex patterns can be applied repeatedly in a number of different ways, each of which is described as a separate variant . For example, Figure 9 shows part of the Composite fragment of our pattern language, including four tiling variants of the Composite pattern. Briefly, a Two-Way Composite describes a graph structure with pointers in both directions, which can be used in dataflow programming [25]; a Cascade is tree of composites where each layer in the tree contains different types of objects [16]; a Two-dimensional composite is a composite where every Component node acts as a Root node in a second composite, as in a tree of heavyweight widgets each containing a tree of lightweight gadgets [15]; and a Lambda Composite involves two superimposed composites, where one composite provides a more abstract view of the second composite, as used in the Trestle window system [24]. Each of these variants both *refines* and *uses* the main Composite pattern.

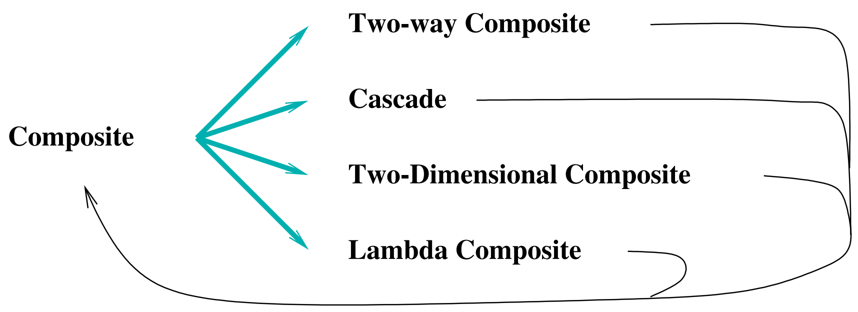

Figure 9: The Composite Fragment

3.4.5 State Fragment

The State fragment incorporates Dyson and Anderson's *State Patterns* pattern language fragment [14] directly into our larger pattern language (see Figure 10). 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 Drive 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 [14].

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.

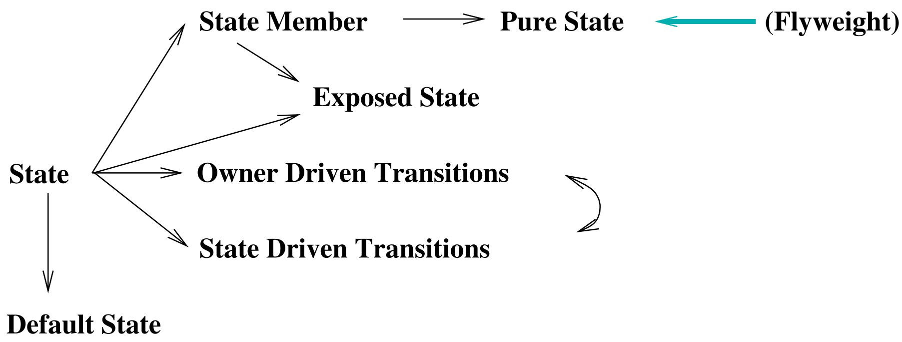

Figure 10: The State Fragment

3.5 The Programming Fragments

The Creation fragment is the most interesting of the Programming fragments, so it is the only one we present here (see Figure 11). This fragment incorporates all the *Design Patterns* creational patterns, the Product Trader pattern [33], and two abstract patterns — Natural Creation and Direct Creation — which introduced to structure the fragment [27]. Natural Creation address the basic question of how objects should be created, and Direct Creation describes the two basic mechanisms for creating objects provided by programming languages — instantiating a class or cloning a prototype.

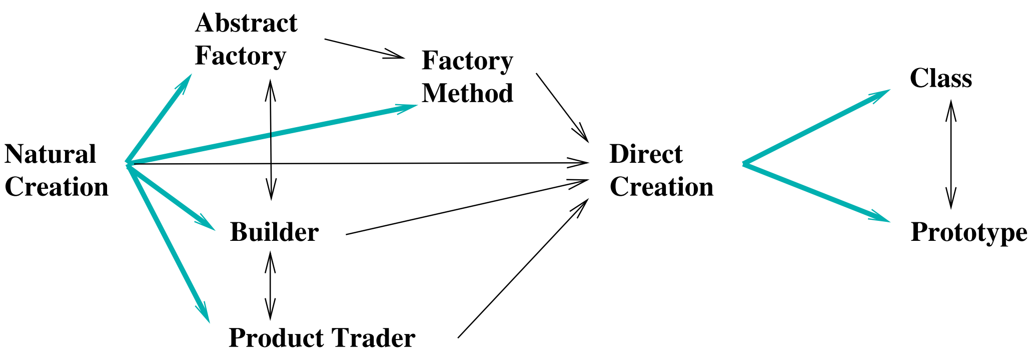

Figure 11: The Creation Fragment

4 Organising A Pattern Language

We used a simple, bottom-up, iterative process to organise patterns into a language. This process had 4 stages:

1. We began by collecting all the patterns we wished to include in the language.

2. We analysed each pattern to determine its relationships with other patterns, and decomposed it into a series of smaller patterns if necessary.

3. We constructed pattern language fragments from the patterns, their sub-patterns, and closely related patterns.

4. We combined the fragments into a whole pattern language.

Generally, we found the first two stages relatively easy. For the first stage, we were able to draw upon a wealth of patterns now available in the literature. Analysing the patterns was also quite straightforward, once we had identified the common kinds of patterns and their characteristic decompositions.

The second two stages, organising patterns into fragments and fragments into languages, were more difficult, and several iterations were required on each fragment at these stages. Indeed, the difficulty of organising all the *Design Patterns* from scratch motivated us to work in two stages, first organising fragments, and then organising a language from the fragments.

At the third stage, some fragments were quite obvious from the source patterns. When we decomposed monolithic patterns with many variants into a main pattern and a number of variant patterns, we would generally place all these patterns into the same fragment — the State fragment is an example of this approach. Fragments built up by including a number of related patterns were more difficult to organise, often requiring us to iteratively introduce (and then analyse) abstract patterns, requiring insight about the underlying structure of the patterns in the fragment — the Shearing fragment is a prime example of this approach.

Similarly, the fourth stage also required us to iterate to find and introduce new patterns to tie the patterns together into the language — including all the high-level patterns in the initial fragment, as the *Design Patterns* do not address low-level analysis and design. We also needed to move patterns between fragments, particularly to organise major patterns and the subpatterns or solution variants which needed to follow them in the language. In this we followed Alexander — inasmuch as *A Pattern Language* decomposes patterns into tightly coupled subpatterns, the subpatterns are placed into a separate fragment following the main pattern.

Finally, at the end of the fourth stage, we produced a linear sequence for the whole language, by sorting the patterns within the fragments into a topological order based on the *uses* relationships between the patterns, and then sorting the fragments based on the aggregate relationships between them. As much as possible, we ensured that larger scale patterns would come before smaller scale patterns, and that patterns appear before the patterns they use, although like most other pattern languages including

Alexander's, we could not avoid backwards references completely.

5 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 [19, 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 patters to start the language. Fortunately, a large number of other general-purpose design patterns have been identified and codified since the publication of *Design Patterns*, and we have been able to organise many of these into the language. As a result, a programmer can begin with the initial OO Program pattern, and traverse through the language to design a program.

The pattern language we have constructed is intentionally initial, partial, and open to extension. Although a complete pattern language, in the sense that a path can be traced through the sequence of patterns to generate a program design, the language still requires many more patterns — in particular, subpatterns for the more complex design patterns, more composite patterns, and more abstract patterns.

6 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 structure of the resulting *Found Objects* pattern language [30], and outlined the contents of the major fragments in the language. We have also described how this language is made up of a variety of kinds of patterns — composite patterns, abstract patterns, problem and solution variants, tiling variants, and subpatterns — and how these kinds of patterns can be identified and organised via their relationships with other patterns. We have also described the way in which we constructed the pattern language — by collecting patterns, analysing the relationships between them, grouping them into fragments and the fragments into a 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.

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, 1996.

[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] Kyle Brown and Bruce G. Whitenack. Crossing chasms, a pattern language for object-RDBMS integration. In Vlissides et al. [35].

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

[9] James O. Coplien. A generative development-process pattern language. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.

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

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

[12] Ward Cunningham. The CHECKS pattern language of information integrity. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.

[13] Ward Cunningham. EPISODES: a pattern language of competitive development. In Vlissides et al. [35].

[14] Paul Dyson and Bruce Anderson. State objects. In Martin et al. [26].

[15] Paula Ferguson and David Brennan. *Motif Reference Manual*. O'Reilly & Associates, Inc., 1993.

[16] Ted Foster and Liping Zhao. Cascade. In *PLOP Proceedings*, 1997.

[17] Martin Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[18] Erich Gamma. Extension object. In Martin et al. [26].

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

[20] Ralph Johnson and Bobby Woolf. Type object. In Martin et al. [26].

[21] Wolfgang Keller. A pattern language for relational databases. Submitted to Europlop'98.

[22] Doug Lea. Christopher alexander: An introduction for object-oriented designers. *ACM Software Engineering Notes*, January 1994.

[23] David H. Lorenz. Tiling design patterns — a case study. In *ECOOP Proceedings*, 1997.

[24] Mark S. Manasse and Greg Nelson. Trestle reference manual. Technical Report 68, DEC Systems Research Center, 1991.

[25] Dragos-Anton Manolescu. A data flow pattern lanuguage. In *PLOP Proceedings*, 1997.

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

[27] Gerard Meszaros and Jim Doble. A pattern language for pattern writing. In Martin et al. [26].

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

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

[30] James Noble. Found objects. Technical report, Microsoft Research Institute, Macquarie University, 1998.

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

[32] Dirk Rielhe. Composite design patterns. In *ECOOP Proceedings*, 1997.

[33] Dirk Rielhe. Product trader. In Martin et al. [26].

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

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

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

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

[38] Bobby Woolf. Null object. In Martin et al. [26].

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