Classifying Relationships Between Object-Oriented Design Patterns

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 object-oriented design patterns have been identified and codified. As part of the pattern form, object-oriented design patterns must indicate their relationships with other patterns, but these relationships are typically described very briefly, and different collections of patterns describe different relationships in different ways. In this paper we describe and classify the common relationships between object oriented design patterns. Practitioners can use these relationships to help them identity those patterns which may be applicable to a particular problem, and pattern writers can use these relationships to help them integrate new patterns into the body of the patterns literature.

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*" [12, p.3]. Designers can incorporate patterns into their program to address general problems in the structure of their programs' designs, in a similar way that algorithms or data structures are incorporated into programs to solve particular computational or storage problems. A growing body of literature catalogues patterns for object-oriented design, including reference texts such as *Design Patterns* [12] or *Patterns of Software Architecture* [5], and patterns compendia such as the *Pattern Languages of Program Design* series [6, 28, 17].

Design patterns do not stand alone — rather each pattern is related other patterns in a variety of ways, and the text of each pattern should describe these relationships. Unfortunately, each pattern text book or catalogue describes relationships between patterns using its own idiomatic classification of these relationships. This inconsistency can make patterns difficult to use, because a pattern's relationships to other patterns is an important part of a pattern's definition, and inconsistent definitions make it difficult for readers to understand the relationships intended by any particular description.

In this paper, we present a classification scheme for the relationships between design patterns. This scheme is based on three *primary* relationships between patterns — a pattern *uses* another pattern, a pattern *refines* another pattern, or a pattern *conflicts* with another pattern. The paper also describes a number of *secondary* relationships between patterns, such as a pattern being *similar* to another pattern, or one pattern *combining* with another pattern, and shows how these secondary relationships can be expressed in terms of the primary relationships.

This paper is organised as follows. Section 2 briefly reviews object-oriented design patterns and the major constituents of the pattern form. Section 3 describes the structure of our classification scheme, and enumerates the relationships we describe. Section 4 then presents the primary relationships in our classification scheme in detail, and Section 5 describes how the secondary relationships between patterns can be expressed in terms of the primary relationships. Section 6 concludes the paper.

2. Object-Oriented Design Patterns

A design pattern is an abstraction from a concrete recurring solution that solves a problem in a certain context [12, 5]. Design patterns were developed by an architect, Christopher Alexander, to describe techniques for town planning, architectural designs, and building construction techniques and described in Alexander's *A Pattern Language • Towns, Building Construction* [1]. Design patterns were first applied to software by Kent Beck and Ward Cunning-

ham to describe user interface design techniques [4], and were then popularised by the *Design Patterns* catalogue, which described twenty-three general purpose patterns for object oriented design. Since *Design Pattern's* publication, a large number of other patterns have been identified and published [5, 3, 2, 6, 28, 17]. More recently, different types of patterns have been identified, including Composite Patterns [25] and Pattern Tilings [14].

A design pattern is written in *pattern form*, that is in one of a family of literary styles designed to make patterns easy to apply. A design pattern has a name to facilitate communication about programs in terms of patterns, a description of the problems for which the pattern is applicable, an analysis of the *forces* addressed by the pattern and the important considerations and consequences of using the pattern, a sample implementation of the pattern's solution, and references to known uses of the pattern.

Most importantly from the perspective of this paper, a pattern lists other patterns which are in some way related. The problem addressed by this paper is that different collections of patterns describe different relationships between patterns in different ways. For example, the relationships between object-oriented design patterns were first analysed by Walter Zimmer, who studied the relationships between patterns in *Design Patterns*[12]. According to Zimmer [31], there are three relationships — one pattern can use another pattern; a variant of one pattern can use another pattern; and one pattern can be similar to another pattern. The *Design Patterns Smalltalk Companion* includes these three relationships, but introduces a fourth relationship — one pattern can be a specialisation of another pattern [2]. *Patterns of Software Architecture* describes three kinds of relationships between patterns, including Zimmer's first relationship and introducing two more — one pattern can be a variant of another pattern; and two patterns can be used in combination to solve a problem [5]. Meszaros and Doble's *Patterns for Pattern Writing* [18] identify five relationships between patterns — a pattern can use, be used by, generalise, specialise, or provide an alternative to another pattern. Other patterns or collections of patterns describe other relationships between patterns and recent work has discussed compositional and reflexive relationships between patterns [27, 25, 14].

We are engaged in a long-term project called "*Found Objects*" that aims to organise patterns for object-oriented design [19, 20, 21, 23]. As part of this project, we have analysed existing patterns to determine their relationships to other patterns. We have identified new patterns, and then related these patterns to the existing patterns in the literature. The classification scheme for relationships between patterns presented in this paper was developed to support the wider goal of organising patterns in the Found Objects project, by identifying the most important relationships between patterns.

3. Classifying Relationships between Patterns

Figure 1 illustrates the structure of our classification scheme for relationships between patterns. The classification scheme is based on three *primary* relationships between patterns — 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. These three relationships are described in Section 4.

The scheme also describes a number of *secondary* relationships between patterns, such as a pattern being a *variant* of another pattern, two patterns being *similar*, or two patterns *combining* to solve a problem. Section 5 presents the secondary relationships, and describes how they can be expressed in terms of the primary relationships.

4. Primary Relationships

Our classification scheme is based on three *primary* relationships between patterns — *uses*, *refines*, and *conflicts*. In this section we outline these three relationships, present examples of well-known patterns which exhibit these relationships, and provide a simple graphical notation which can illustrate these relationships between groups of patterns.

4.1. Uses

Almost every pattern form records the patterns which each pattern *uses* — that is, patterns which help another pattern. Typically, a pattern which has a *larger* or more global impact on a design will use patterns which have *smaller* or more local impacts.

The *uses* relationship is the only explicit relationship between patterns in Alexander's original architectural patterns [1]. Alexander's patterns are organised into a *pattern language* based on this relationship — that is, the patterns are organised so that each pattern is followed by the other smaller-scale patterns which it uses, and which should be considered once the large-scale pattern has been chosen. For example, the Small Public Squares pattern *uses* the Pedestrian Density, Activity Pockets, and Something Roughly in

Primary Relationships	
Uses	One pattern uses another pattern [1, 12, 31]
Refines	A specific pattern refines a general pattern [2, 18]
Conflicts	A pattern addresses the same problem as another pattern [8, 3]
Secondary Relationships	
Used by	A smaller pattern is used by a larger pattern [1]
Refined by	A general pattern is refined by a specific pattern [18]
Variant	A variant pattern refines a more well-known pattern [5]
Variant Uses	A variant of one pattern uses another pattern [31]
Similar	A pattern is similar to another pattern [31]
Combines	Two patterns combine to solve a single problem [5, 8]
Requires	A pattern requires the solution of another pattern [7]
Tiling	A pattern uses itself [14]
Sequence of Elaboration	A sequence of patterns from the simple to the complex [26, 11]

Figure 1. The Structure of the Classification Scheme

the Middle patterns. The Small Public Squares pattern is a larger scale pattern because it describes the design of the entire square; the other patterns are smaller because they describe the pedestrian traffic in the square, the edges of the square, and the centre of the square respectively.

Software pattern forms explicitly record this relationship — often mixed with others — typically in a section titled *Related Patterns* [12] or *See Also* [5]. For example, the Observer pattern that maintains consistency between multiple views of objects *uses* the Mediator pattern to coordinate updates of multiple objects. In turn, the Mediator pattern *uses* the Singleton pattern to ensure that Mediators are not duplicated unnecessarily. This is most important relationship in Zimmer's analysis of the relationships between design patterns [31] and is explicitly represented in every published collection of design patterns [6, 28, 17].

The *uses* relationship can be used to simplify the descriptions of more complex patterns by composition. Riehle has defined a *composite* design pattern as "*any design pattern which is best described as the composition of further patterns*" [25]. For example, Riehle describes how the Model-View-Controller pattern [5] can be seen as a composite of the Observer, Strategy, and Composite patterns: the Observer pattern connects a View to its Model, the Strategy pattern arranges for a Controller object to handle input on behalf of a View, and the Composite pattern provides a hierarchy of Views. Similarly, Dyson and Anderson have written a collection of small patterns which describe in detail how to apply the State pattern [8].

The important point here is that a composite pattern *uses* the smaller-scale patterns of which it is composed. The description of a composite pattern must describe *how* these patterns are combined into the composite pattern, for example, by using role diagrams [25]. Following Zimmer [31], Figure 2 shows how the structure of *uses* relationships can be illustrated graphically.

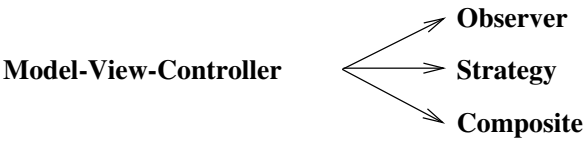

Figure 2. The *uses* relationship in the Model-View-Controller Pattern

4.2. Refines

The second primary relationship in our classification scheme is that one pattern *refines* another pattern, that is, one pattern is a specialisation of a more general, more simple, or more abstract pattern.

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.

For example, in *A Pattern Language* the Sequence

of Sitting Spaces pattern *refines* the Intimacy Gradient Pattern [1], even though this is not identified explicitly in the text. The Intimacy Gradient pattern describes how a house should be designed so that the public rooms are closest to the main entrance, while the private rooms are furthest from the door, while the Sequence of Sitting Spaces pattern describes how chairs, alcoves and window seats should be designed to form an Intimacy Gradient.

This refinement relationship is also found in software patterns. For example, in *Design Patterns* the Factory Method pattern *refines* the Hook Method pattern [12]. Hook Methods allow subclasses to override parts of the behaviour of Template Methods defined in superclasses. One force addressed by the Hook Method pattern is the desirability of naming conventions to make clear to later programmers that a particular method is in actually a Hook Method. Factory Method *refines* Template Method, because Factory Methods are effectively Hook Methods which are used by subclasses to specify the class of an object the Template Method in the superclass will create. In the description of the Factory Method pattern, one of the main forces addressed by the pattern is the use of naming conventions to illustrate the a particular method is in fact a Factory Method.

The *refines* relationship can be used to identify new *abstract patterns* by generalisation — existing patterns which have structure in common can be seen as refinements of an abstract pattern which factors out the commonalities. For example, the Iterator pattern [12], the Type-safe Session pattern [24], the Accumulator pattern [30] and the Pen objects used in many graphic systems [13] all refine the Curried Object abstract pattern [20]. Figure 3 illustrates this refinement relationship for the Curried Object pattern.

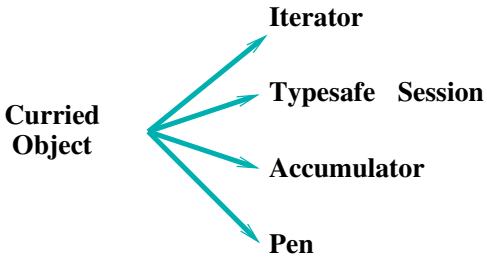

Figure 3. The *refines* relationship in the Curried Object Pattern

The Curried Object pattern describes how one *client* object can provide a simplified interface to another *server* object by using local state to store information about the client's use of the server object, and adding this information into the messages it receives before forwarding them on to the Server. Conceptually, an Iterator provides an interface to a collection by managing a "current position" in the collection, and when the iterator is used, the current position is passed into the collection along with any other arguments to the message. Similarly, a Pen maintains a "current position" on a graphic medium which represents the position of the graphics pen, and the Accumulator maintains the "current arguments" to be used to initialise the server object.

The *Design Patterns Smalltalk Companion* [2] also introduces a number of abstract patterns such as Sharing and OO Recursion, and describes other patterns as refinements of these abstract patterns. For example, Flyweight *refines* Sharing, and Chain of Responsibility *refines* OO Recursion.

4.3. Conflicts

The third fundamental relationship between patterns in our classification scheme is that they *conflict* — that is, that two or more patterns provide mutually exclusive solutions to similar problems. 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.

For example, Kent Beck's *Smalltalk Best Practice Patterns* [3] includes an excellent example of two conflicting patterns which both address the problem of how a Smalltalk programmer should access an object's instance variables. The Direct Variable Access pattern presents one solution — that instances variables should be accessed directly, because this is easy to read and compiles efficiently. In contrast, the Indirect Variable Access pattern presents a diametrically opposed solution — that instances variables should only be accessed indirectly, via accessor methods, because this makes it easy to change the representation of the variable without affecting code that uses it. Figure 4 illustrates this relationship using the same notation as Dyson and Anderson [8].

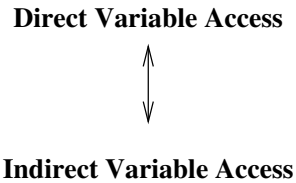

Figure 4. The *conflicts* relationship in the Variable Patterns

Considering patterns from *Design Patterns*, the

Decorator pattern conflicts with the Strategy pattern in that both patterns can be used to modify the behaviour of other objects — for example, both patterns have been used to add graphical borders or icons to window objects in different window system frameworks [12, p.180]. The Prototype and Factory Method patterns also conflict, because they provide two alternative solutions to the problem of subclasses redefining the classes of objects created in superclasses.

When reading or applying a pattern language, this relationship can be exploited in two ways — when looking for patterns, if a pattern seems as if it may be applicable, then the conflicting patterns should be examined because they present alternative choices, but once one pattern has been chosen, the other conflicting patterns can be ignored.

It is interesting to note that Alexander's pattern language does not use this relationship explicitly, indeed, Alexander claims every pattern can be used with any other pattern [1]. In practice, Alexander's pattern language includes a number of conflicting patterns, for example the House for a Small Family, House for a Couple, and House for One Person patterns describe three mutually exclusive kinds of houses.

5. Secondary Relationships

In describing the relationships between patterns, pattern writers often use a number of other relationships between patterns. We classify these relationships as *secondary* relationships, because we have been able to express them in terms of the primary relationships. In this section we present the secondary relationships we have classified. Some of these relationships, such as *used by* and *refined by*, are simply inverses of primary relationships, but other relationships, such as *variant of* and *combines with*, are more complex.

5.1. Used By

The *used by* relationship is the inverse of the *uses* relationship, and can be analysed in the same way as that relationship. For example, because Mediator *uses* Singleton, Singleton is used by Mediator [12]. Similarly, Iterator is used by Interpreter and Visitor.

Some pattern forms, including Alexander's, explicitly include a section to record the used by relationship, to give the context of more general patterns within which a particular pattern is likely to be instantiated. Other pattern forms occasionally record this relationship within the related patterns sections or omit it all together.

5.2. Refined By

The *refined by* or *generalises* relationship is the inverse of the *refines* relationship, and can be analysed in the same way as that relationship. For example, if Factory Method refines Template Method, then Template Method is refined by Factory Method, and so on.

Most existing pattern forms only record the refined by relationship occasionally, if at all.

5.3. Variants

A pattern is a recurring solution to a design problem. Because every problem is different, a pattern will be instantiated differently every time it appears. In practice, some kinds of problems and solutions will occur more regularly than others, so some ways of instantiating patterns are more common in practice than others. These common patterns of instantiation are called pattern *variants*, and some pattern forms, in particular *Patterns of Software Architecture*, explicitly identify pattern variants, either as separate patterns or as parts of other main patterns [5].

We decompose the variant relationship by treating each variant as a separate pattern. Not all variations are sufficiently important to be identified explicitly as variants in the language — but where variants are dignified with a particular name, or are substantially different from the main pattern, that is they extend the main pattern's structure or make different tradeoffs between forces, and , then they are worth documenting as variant patterns in their own right.

This decomposition makes clear that the variant relationship is used in two distinct ways in the patterns literature. Most pattern variants are *solution variants*, that is, they provide alternative solutions to the same problem, as described above. A few pattern variants are *problem variants*, that is, they describe how a single pattern can provide a solution to a number of different problems. We analyse the relationships between problem and solution variants differently, as follows.

5.3.1 Solution Variants

Most pattern variants describe variant solutions to a common problem. For example, the Adaptor pattern has two main variant solutions — a Class Adaptor and an Object Adaptor [12]. The Adaptor pattern *"converts the interface of an* [*Adaptee*] *class into another* [*existing incompatible Target*] *interface clients expect. Adaptor lets classes work together that couldn't otherwise, because of incompatible interfaces"* (*Design*

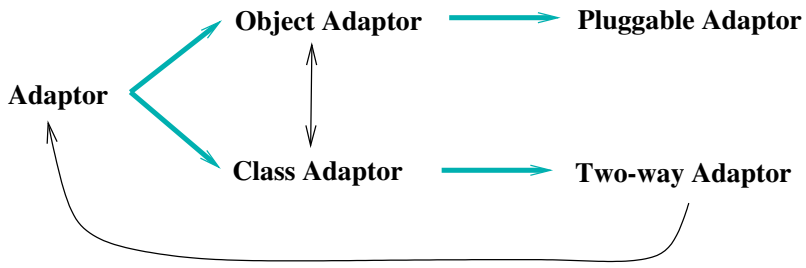

Figure 5. Adaptor pattern variants

Patterns [12, p. 139]). In this way, the Adaptee class can be used as a server by client objects which access it through the Target interface, without requiring modification of the client objects. The two variants of the adaptor problem provide two solutions to this problem: a class adaptor extends the Adaptee class so that it implements both its own interface and the Target interface, typically by using multiple inheritance, while an object adaptor introduces a new object which implements the Target interface by delegating to the Adaptee. Class and Object Adaptors trade off forces in different ways — for example, a class adaptor can change the interface of only one concrete Adaptee class, while an object adaptor can adapt an Adaptee class and all of its subclasses. In the *Design Patterns* book, these two adaptor variants are described within the text of the overall adaptor pattern, but have individual structure diagrams, consequences, implementation issues, and sample code.

The *Design Patterns* book also describes another two variants of the Adaptor pattern, although in less detail than the Class Adaptor and Object Adaptor. A Pluggable Adaptor builds interface adaption into the Target object so that it can be configured to support a range of Adaptee interfaces, and a Two-way Adaptor is a Class Adaptor which adapts two Adaptee interfaces to each other — that is, the two interfaces are simultaneously play the roles of the Target and Adaptee interfaces in the Class Adaptor pattern.

We decompose the relationship between solution variant patterns into the *refines* and *conflicts* relationships. We treat a solution variant as a separate pattern which *refines* the main pattern and *conflicts* with the other variants. A solution variant pattern *refines* the main pattern because solution variants provide specialised solutions to the problem addressed by the main pattern, and *conflicts* with other variants because solution variants are usually mutually exclusive.

Figure 5 illustrates the relationships in a pattern language fragment containing the Adaptor pattern and its variants. The main Adaptor pattern introduces the common problem — adapting the interface of an object. The four solution variants are refinements of the main Adaptor pattern, linked by the *refines* relationship — a Class Adaptor is a more specific solution to the general problem of interface adaption, as is an Object Adaptor, and a Pluggable Adaptor is a more specific version of an Object Adaptor. Class and Object Adaptor are conflicting patterns, because they are mutually exclusive solutions to any given adaption problem. The Two-way Adaptor also *uses* the main Adaptor pattern: this is discussed in Section 5.8 below.

5.3.2 Problem Variants

Although most pattern variants describe variant solutions to a common problem, some pattern variants describe variant problems which share a common solution. For example, the Proxy pattern is described both in *Design Patterns* and *Patterns of Software Architecture*, and each description introduces a number of variants — four in *Design Patterns*, and these four plus another three in *Patterns of Software Architecture*. The Proxy pattern is used to "*Provide a surrogate or placeholder for another object* [*the Subject*] *to control access to it*" (*Design Patterns* [12, p. 207]). A Proxy object provides the same interface as the original Subject object, but intercepts any messages directed to the Subject. A Proxy object can therefore be used in place of the Subject by a client which are designed to access the Subject, without the client being aware the Subject has been replaced by a Proxy.

Because a Proxy intercepts any message directed to its Subject, a Proxy can handle or override these message. The main Proxy pattern describes how these messages can be intercepted — that is, it describes a solution — but it does not describe a single problem which this solution resolves. Rather, the many variants of the Proxy pattern each describe a different problem to which Proxy provides the solution. For example, a Virtual Proxy creates expensive objects on demand so that they will only be created if necessary, a Remote Proxy is a local representative for an object on a remote machine so that other local clients can ac-

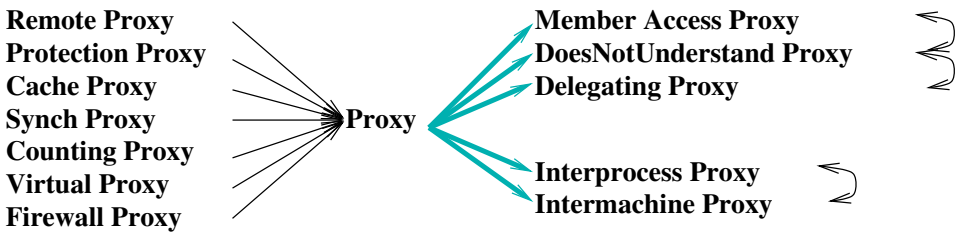

Figure 6. Proxy pattern variants

cess the object without knowing about details of interprocess communication, and a Protection Proxy provides a level of security by checking the access rights of the client sending messages to the proxy. *Patterns of Software Architecture* makes this 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*" [5, p. 263].

As with the Adaptor pattern, we decompose the Proxy pattern into a number of related patterns, forming a pattern language fragment [23]. The main Proxy pattern introduces the common solution — providing a placeholder for an object. Each problem variant is then captured as a separate pattern which *uses* the main Proxy pattern. The main pattern *uses* rather than *refines* the variants because the main pattern is a common solution employed by each variant, rather than a specialisation of every variant. We do not need to record a *conflicts* relationship between different problem variants, because these variants all address different problems.

In fact, Proxy also has a number of *solution* variants, so the relationships between the various Proxy pattern variants are more complex than this. Figure 6 shows the overall structure of the Proxy fragment. Each solution variant *refines* the main Proxy pattern and *conflicts* with mutually exclusive variants.

5.4. Variant Uses

Zimmer [31] distinguishes between two forms of the *uses* relationship, "*X uses Y in its solution*" — where all instantiations of pattern X use pattern Y — and "*Variant of X uses Y in its solution*" — where some instantiations of pattern X will use pattern Y, and others will not. For example, an instantiation of the Visitor pattern will always use the Double-Dispatch pattern, but may or may not use the Iterator pattern — in Zimmer's terms, a variant of Visitor would use Iterator. Similarly, a variant of Composite may use Visitor or Iterator [31].

Our pattern classification scheme does not make this distinction, rather, we use only the primary *uses* relationship. There are several reasons why we handle variant uses in this way. The first reason is philosophical — since patterns are ultimately guidelines or rules of thumb, and are meant to be tailored to suit a specific problem and the context of their use, any part of a pattern may be omitted or varied when the pattern is instantiated [1, 12]. The second reason is that we identify the important pattern variants explicitly, treating them as first class patterns as described in Section 5.3 above, so that this relationship can be classified as a full *uses* relationship with the variant pattern, rather than a *variant uses* relationship with the main pattern, assuming the variation is significant. The third reason is pragmatic — we are interested in classifying the relationships between patterns to help us organise patterns, and the *uses* relationship must be considered whether the use is mandatory or optional. Treating the two relationships in common when organising patterns into a pattern language or system does not preclude the text of individual patterns from making finer distinctions.

5.5. Similarity

Zimmer also introduces a similarity relationship between patterns, "*X is similar to Y*" [31]. This relationship is derived from comments in the related patterns section of *Design Patterns*, and it seems to be a catch-all category for relationships between patterns which are not captured by the other relationships.

This relationship is often used to describe patterns which are similar because they address the same problem, that is, some of the patterns Zimmer would classify as being similar we would classify as conflicting, such as Abstract Factory, Prototype, and Builder. The similarity relationship seems to be much broader that just *conflicts*, however, as it is also used to describe patterns which have a similar solution technique, such as Strategy and State, or which are occasionally used together, such as Flyweight and Strategy.

We express these kind of relationships using more specific arrangements of primary relationships as ap-

propriate. For example, patterns which have similar solution techniques can be treated as solution variants, that is, as refining a more abstract pattern (see 5.3.1), and patterns which are used together can be related by the primary *uses* relationship.

5.6. Combines

Patterns of Software Architecture [5] and Paul Dyson's *Patterns for Abstract Design* [7] introduce a *combines* relationship between patterns, where two patterns combine to solve a single problem which is not addressed directly by any other pattern. For example, from *Patterns of Software Architecture*, the Proxy and Forwarder-Receiver patterns can be combined to implement a transparent peer-to-peer communication service [5, p. 17]; from *Patterns for Abstract Design*, the Domain Abstraction Classes and Framework Implementation classes patterns can be combined to implement a software framework, and the Abstract Mechanistic Behaviour and Abstract Significant Roles patterns can be combined to implement an abstract design.

In simple cases, we can model this relationship directly by the *uses* primary relationship, where one pattern is a larger scale pattern that addresses the whole problem and the other pattern is a smaller scale pattern which provides a solution to a subproblem. For example, in the Proxy and Forwarder-Receiver pattern combination, the Forwarder-Receiver pattern describes how to implement a communications substrate which is used by the Proxy pattern. We classify these patterns so that the Proxy pattern *uses* the Forwarder-Receiver pattern.

In more complex cases, such as the two other examples above, we consider that this relationship really points to a lack in the patterns themselves — although these patterns combine to provide a solution to a problem, the actual problem and the way these patterns combine to solve it is being represented by the combines relationship, and is not captured explicitly in a pattern. In these cases, we ensure that the problem is identified explicitly by locating an existing pattern or introducing a new pattern which addresses the problem directly, outlines the whole solution, and *uses* the patterns that combine to solve it.

For example, the Domain Abstraction Classes and Framework Implementation classes patterns combine to implement a software framework. The *Patterns for Abstract Design* include a Software Framework pattern which *uses* the other two patterns, so the combines relationship is already expressed by the *uses* relationship here. The Abstract Mechanistic Behaviour and Abstract Significant Roles patterns combine to implement an abstract design, but Dyson does not include an "Abstract Design" pattern. To model this relationship, we would introduce Abstract Design as an explicit pattern which *uses* the Abstract Mechanistic Behaviour and Abstract Significant Roles patterns.

The underlying intuition here is that patterns use a number of other patterns to provide the solution to the problem they address. All patterns can be considered to combine with every other pattern which is used-by another (often larger-scale) which pattern *uses* them, to solve the problem the larger-scale pattern addresses.

5.7. Requires

Patterns for Abstract Design also introduces a *requires* relationship between patterns — one pattern *requires* a second pattern if the second pattern is a prerequisite for solving the problem addressed by the first pattern [7]. For example, the Software Framework, Abstract Significant Roles and Abstract Mechanistic Behaviour patterns all require the Defined Domain pattern — that is, the Defined Domain pattern must have been applied before the other patterns can be applied successfully.

In general, we consider that this relationship can be modelled quite adequately by the *uses* primary relationship. The distinction between *requires* and *uses* seems to be based primarily on the order in which the patterns should be applied — if one pattern requires a second pattern, the second pattern must be applied before the first pattern can be used to produce its solution. This is also the case with the general *uses* relationship, since if one pattern *uses* a second pattern, the second pattern must be applied before the solution described by the first pattern will be complete. For this reason, we model the requires relationship with the primary *uses* relationship.

5.8. Tiling

Some patterns can be applied repeatedly to solve a single problem. Lorenz has identified some particular examples of this as *Pattern Tilings* [14] — for example, Interpreter and Visitor patterns can be instantiated repeatedly to implement a reflexive system. Other patterns can also be tiled — for example, an Iterator can iterate over other iterators [29], and the Two-Way Adaptor pattern variant described in Section 5.3.1 above can be seen as a tiling of the Adaptor pattern, because the Class Adaptor pattern is applied twice, once for each direction.

We express pattern tiling by introducing additional variants of the patterns which are applied reflexively.

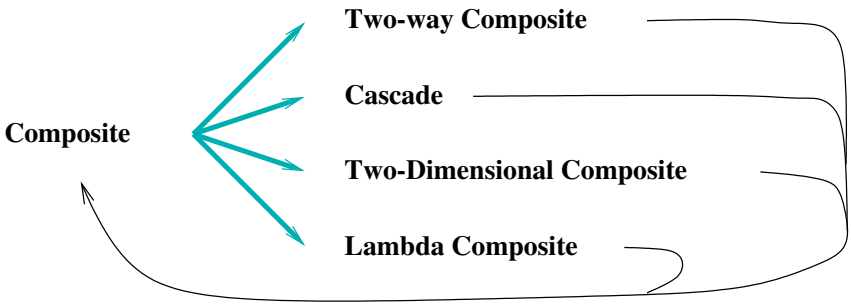

Figure 7. Composite pattern tiling variants

As with more typical solution variants, a reflexive variant *refines* the main pattern, however a reflexive variant also *uses* the main pattern. With this approach, a pattern which can be tiled is simply one which *uses* itself, and pattern tiling does not need to be treated as a "*fundamental reflexive relationship*" within the pattern language [14].

Introducing explicit tiling variant patterns can also make clear that complex patterns can be tiled in a number of different ways. For example, Figure 7 shows that the Composite pattern can be tiled in four different ways. Briefly, a Two-Way Composite describes a graph structure with pointers in both directions, which can be used in dataflow programming [16]; a Cascade is tree of composites where each layer in the tree contains different types of objects [10]; 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 [9]; 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 [15]. Each of these variants both *refines* and *uses* the main Composite pattern.

5.9. Sequences of Elaboration

We recognise a *sequence of elaboration* as a sequence of patterns, which begins with small, simple, low-level patterns, with few negative consequences, and proceeds through a number of patterns of increasing complexity to large scale architectural patterns which have large negative consequences for the complexity and performance of the system. Often a standalone collection of patterns or a fragment of a pattern language will form a sequence of elaboration.

For example, the *Do-It-Yourself Reflection* pattern collection [26] contains a sequence of patterns from the small, local Property List pattern which affects a single object, via the Anything pattern to the larger scale Configured Object Registry. The Object System pattern [22] completes this sequence by describing how Do-It-Yourself reflection can be used to define the architecture for an entire system. Many chapters in Fowler's *Analysis Patterns* are structured as sequences of elaboration [11].

Sequences of elaboration are quite adequately described by the primary relationships between the patterns in the sequence — typically, the smaller patterns in the sequence are refined by the larger patterns, and the larger patterns *use* the smaller patterns in the sequence. These relationships imply opposite orderings on the patterns when the sequences (see Figure 8).

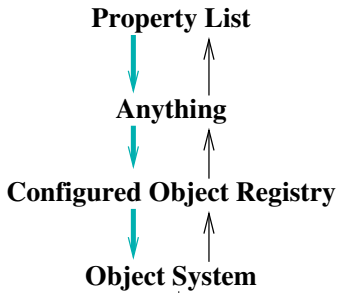

Figure 8. Sequence of Elaboration

The *uses* relationship is typically ordered from a composite pattern to a pattern which uses it — in sequences of elaboration, from larger patterns to smaller patterns. In contrast, the refinement relationship is typically ordered from more general patterns to more specialised patterns — in sequences of elaboration, this implies that the more general and more applicable small scale patterns should be presented first. Based on our experience, and on the sequences of elaboration we have found in the literature, we consider that sequences of elaboration should treat the *refines* relationship as more important than the *uses* relationship, and so sequences of elaboration should be presented

beginning with the simple patterns, and then proceeding on to the more complex patterns.

6. Conclusion

In this paper, we have described how the relationships between object-oriented design patterns can be classified. Our classification scheme is based on three *primary* relationships between patterns — a large scale pattern *uses* a small scale pattern, a more specialised *refines* a more general pattern, and a pattern *conflicts* with other patterns which provide different solutions to the same problem. A variety of *secondary* relationships between patterns — *variant uses*, *similarity*, *combination*, *generalisation*, and so on — can then be expressed in terms of the primary relationships, sometimes requiring patterns to be decomposed into variants or the introduction of extra higher level patterns.

This classification scheme can be used to analyse existing patterns, to organise them based on the analysis, and to improve new patterns which are being written. First, existing patterns can be inspected, and their interrelationships described using our classification scheme — this paper provides numerous examples of such analysis. Second, the patterns themselves can then be catalogued or classified based on their relationships to other patterns — for example, we are currently working on classifying the patterns from *Design Patterns* into a pattern language based on the patterns' relationships [23]. Finally, if the relationship of a new pattern to other patterns are described in terms of this classification scheme, it should be easier for readers of the new pattern to understand that pattern's relationships to other new or existing patterns.

Acknowledgements

Thanks to Geoffrey Outhred who suggested the idea for this paper, and kindly commented on drafts.

References

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

[2] S. R. Alpert, K. Brown, and B. Woolf. *The Design Patterns Smalltalk Companion*. Addison-Wesley, 1988.

[3] K. Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1996.

[4] K. Beck and W. 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.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[6] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1996.

[7] P. Dyson. *Patterns for Abstract Design*. PhD thesis, University of Essex, 1997.

[8] P. Dyson and B. Anderson. State objects. In Martin et al. [17].

[9] P. Ferguson and D. Brennan. *Motif Reference Manual*. O'Reilly & Associates, Inc., 1993.

[10] T. Foster and L. Zhao. Cascade. In *PLOP Proceedings*, 1997.

[11] M. Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[12] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[13] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

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

[15] M. S. Manasse and G. Nelson. Trestle reference manual. Technical Report 68, DEC Systems Research Center, 1991.

[16] D.-A. Manolescu. A data flow pattern lanuguage. In *PLOP Proceedings*, 1997.

[17] R. C. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1998.

[18] G. Meszaros and J. Doble. A pattern language for pattern writing. In Martin et al. [17].

[19] J. Noble. Found objects. In *EuroPLOP Proceedings*, 1996.

[20] J. Noble. Arguments and results. In *PLOP Proceedings*, 1997.

[21] J. Noble. Basic relationship patterns. In *EuroPLOP Proceedings*, 1997.

[22] J. Noble. The object system pattern. In *EuroPLOP Proceedings*, 1998.

[23] J. Noble. Organising patterns into languages: Towards a pattern language for object oriented design. Submitted for publication, 1998.

[24] N. Pryce. Type-safe session. In *EuroPLOP Proceedings*, 1997.

[25] D. Riehle. Composite design patterns. In *ECOOP Proceedings*, 1997.

[26] P. Sommerlad and M. Rüedi. Do-it-yourself reflection. In *EuroPLOP Proceedings*, 1998.

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

[28] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[29] S. A. Yeates. Design of a garbage collector using design patterns. In *TOOLS Pacific 25*, 1997.

[30] P. M. Yelland. Creating host compliance in a portable framework: A study in the use of existing design patterns. In *OOPSLA Proceedings*, 1996.

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