



Patterns as Signs: A Semiotics of Object-Oriented Design Patterns

James Noble ^a Robert Biddle ^b and Ewan Tempero ^c

^a School of Mathematics, Statistics and Computer Science,
Victoria University of Wellington, New Zealand

^b Human Oriented Technology Lab, Carleton University, Canada

^c Department of Computer Science, University of Auckland, New Zealand

Abstract

Object-oriented design patterns have been one of the most important and successful ideas in software design over the last ten years, and have been well adopted both in industry and academia. We provide a semiotic account of design patterns, treating a pattern as a sign comprised of the programmers' intent and its realisation in the program. A number of open research problems remain regarding patterns, including the differences between patterns, variant forms of common patterns, the naming of patterns, the organisation of collections of patterns, the relationships between patterns, and the extent to which patterns modify the rhetoric of object-oriented design. Considering patterns as signs can address many of these common questions regarding design patterns, to assist both programmers using patterns and authors writing them.

Keywords: software design, object-orientation, patterns, semiotics

Earlier versions of parts of this article were presented at the European Conference on Object-Oriented Programming (Noble and Biddle, 2002) and the Australian Computer Science Conference (Noble et al., 2002).

1 Introduction

An object-oriented design pattern is a “*description of communicating objects and classes that are customised to solve a general design problem in a particular context*” (Gamma et al., 1994, p.3). Software 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* (Gamma et al., 1994) or *Pattern-Oriented Software Architecture* (Buschmann et al., 1996, Schmidt et al., 2000), and patterns compendia such as the *Pattern Languages of Program Design* series (Coplien and Schmidt, 1995, Vlissides et al., 1996, Martin et al., 1998, Harrison et al., 2000).

Unfortunately, there are a number of important open research problems regarding patterns. These include: what are the differences between outwardly similar patterns (such as Strategy and State); how can one pattern solve more than one problem (such as Proxy); have distinctly different variant forms (such as Adaptor); how can several different patterns have the same name (such as Prototype); and how can the relationships between patterns best be characterised.

In this article, we provide a semiotic account of design patterns. Semiotics is the study of signs in society that investigates the way meaning is carried by communication, treating communication as an exchange of signs. When semiotics began in the early years of the last century, most work was concerned with conventional signs — first speech, and then writing (Eco, 1976). Since then, the scope of semiotics has widened to cover all kinds of signs, to the point where semiotics underlies much of structuralist and post-structuralist literary theory, film studies, cultural studies, advertising, and even the theory of popular music and studies of communications between animals (zoosemiotics) and within them (biosemiotics) (Sebeok, 2001). One of the avowed values of the design patterns movement is to treat “patterns as literature” (Kerth and Cunningham, 1997, Coplien, 1997); our semiotic approach builds on this idea by applying techniques from the study of literature and culture to programs and patterns.

This article brings together and unifies our work on the semiotics of design patterns (Noble and Biddle, 2002, Noble et al., 2002). The article is organised as follows. Section 2 briefly reviews object-oriented design patterns and the major constituents of the pattern form, and section 3 provides a brief introduction to semiotics and the structure of signs. Next, section 4 presents our semiotic model of design patterns, section 5 addresses a number of open questions in the analysis of design patterns, and section 6 considers a semiotic account of the relationships between patterns. Section 7 then discusses the extent to which patterns affect the rhetoric of object-oriented design, while section 8 considers the wider ramifications of our approach. Section 9 places our approach in the context of other work organising and theorising about patterns, and other work on the semiotics of information processing. Finally, section 10 draws out some possible future directions for a semiotic approach.

2 Object-Oriented Design Patterns

A pattern is an abstraction from a concrete recurring solution that solves a problem in a certain context (Gamma et al., 1994, Buschmann et al., 1996). Patterns were developed by an architect, Christopher Alexander (Lea, 1994), to describe techniques for town planning, architectural designs, and building construction techniques, and are described in Alexander’s *A Pattern Language ? Towns, Buildings, Construction* (Alexander, 1979, Alexander, 1977, Alexander, 1999). Design patterns were first applied to software by Kent Beck and Ward Cunningham to describe user interface design techniques (Beck and Cunningham, 1987, Kerth and Cunningham, 1997), and were then popularised by the *Design Patterns* catalogue, which described twenty-three patterns for general purpose object-oriented design. Since *Design Patterns*’ publication, a large number of other patterns have been identified and published. More recently, different types of patterns have been identified, including Composite or Compound Patterns (Riehle, 1997a, Vlissides, 1998). Although our approach is applicable across the range of software patterns, in this article we take our examples

from Gamma et. al.'s *Design Patterns* (Gamma et al., 1994) because this is the best known collection of patterns.

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 (Coplien, 1996, Meszaros and Doble, 1998, Riehle and Züllighoven, 1996). 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* (important concerns) addressed by the pattern, 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 and to other patterns to which it is related.

More so than other forms of writing about software, patterns are self-consciously "*literature*" about software. The "Pattern Languages of Program Design" (or "PLoP") conference series, for example, has modelled itself on some practices of the creative writing community. At PLoP conferences, for example, papers are workshopped to improve their *expression* (as against their *content*), rather than being presented to a passive audience (Coplien, 2000b). The patterns movement catchphrase "*the aggressive disregard for innovation*"¹ encapsulates this idea: the focus is on the literary expression of existing tested ideas, rather than the advocacy of new idiosyncrasies.

The patterns movement's focus on literature has partly inspired our interest in applying semiotics to patterns. Semiotics is the foundation of structuralist and post-structuralist literary theory, so if patterns are indeed literature, and a *critical* literature in particular, they should be amenable to study using the same tools as other forms of literature or culture.

3 Semiotics

Semiotics, as defined by Saussure (de Saussure, 1916), is the study of signs in society; where a *sign* is "*something standing for something else*" (Eco, 1976). Since Saussure, semiotics has been applied to a wide range of different kind of signs, and for a range of diverse purposes (Andersen, 1997, Cobley and Jansz, 1997, Easthope and McGowan, 1992, Eco, 1976, Edgar and Sedgwick, 1999, Cobley, 2001).

The key idea underlying Saussurean semiotics is a binary model of the *sign*, shown in Fig. 1.

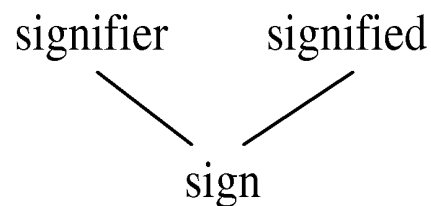


Figure 1: Saussure's Sign

¹ Attributed to Thomas J. "Tad" Peckish by Brian Foote (Foote, 1998).

A sign is a two-part relationship between a *signifier* and a *signified* – a computer scientist might write “sign = signifier + signified”. The signifier (or *expression* of the sign) is some phenomenon that an individual can see, hear, sense, or imagine; and the signified (or *content*) is the mental concept that the signifier produces. For example, consider the English colour name purple as a sign. The spoken or written word “purple” is the signifier while the resulting concept of the colour purple is the signified.

Following Charles Peirce, American semiotics takes a sign as a three-part relationship, including an *object* or *referent*, as well as the signifier (called a *representamen*) and signified (*interpretant*). We use Saussure’s binary sign in the first part of this article as it suffices for our analysis (Eco, 1976), however we will employ Peircean semiotics in section 7 when we consider the question of reference in object-oriented design.

Although semiotics has been employed to study many areas of cultural practice from high culture to comics, there has been surprisingly little work in the direct application of semiotics to computer science. Peter Bøgh Andersen has completed the most work in this area, establishing a sub-field of Computer Semiotics focusing on human-computer interaction and the programming required to support user interfaces and pervasive computing, but also addressing a broader background (Andersen, 1997, Andersen et al., 1997, Andersen and Nowack, 2002, Andersen, 2003). Andersen also argued for a semiotic approach to information systems, rather than relying solely upon generative grammars or logic (Andersen, 1992). Goguen has established Algebraic Semiotics, also primarily concerned with user interface design (Goguen, 1999) and design notations (Goguen, 1993), focusing on formal systems, and Stamper and others have developed an explicitly semiotic methodology for information systems development (Liu et al., 2002, Liu, 2000). Separately, we have also used semiotics to analyse the use of metaphor in user interface design and programming (see e.g. (Barr, 2003, Noble et al., 2002)). In this article, however, we are not aiming to establish new methodologies or programming paradigms. Rather, we present a semiotic analysis of the use of design patterns in practice.

4 Patterns as Signs

In the classic definition, a pattern is a “*solution to a problem in a context*” (Lea, 1994, Hillside Inc., 2001). An object-oriented design pattern, for example, is a description of a piece of knowledge about object-oriented programming or design phrased as a solution to a problem; an architectural pattern (as in *A Pattern Language* (Alexander, 1977)) is a description of a piece of knowledge about architectural design. We call the descriptive part of a pattern a *pattern-description*. Patterns have a secondary function (emphasised more by *Design Patterns* than Alexander) of providing a working vocabulary with which designers can communicate. This section begins by modelling pattern descriptions as signs, and then considers how those pattern descriptions are named.

4.1 Pattern Descriptions

The *solution* is the core of a pattern description. An average pattern in *Design Patterns* is about ten pages long, and eight of these pages are taken up with a description of the solution of the pattern. This description is quite concrete: it is both graphical (using class and sequence diagrams) and textual (with descriptions of participants in

the pattern, possible implementations, annotated example source code, and descriptions of known uses). In a program that uses the pattern, the elements corresponding to the pattern's solution can literally be pointed to in a listing of the program's source code or on a diagram showing the program's classes — a pattern describes a *type* of solution, and a particular solution embodied in a program is a *token* of that type.

A similarly concrete solution is also at the core of each of Alexander's architectural patterns: the elements of the pattern's solution can literally be touched inside a building that incorporates the pattern, or pointed out on the building's plan. Alexander insists that each pattern should be accompanied by a sketch, diagram, or photograph, presumably to ensure the pattern describes a concrete solution.

Note that although the description of a pattern's solution must be concrete, capable of being incorporated into a program or building, this incorporation is not necessarily straightforward — just as the same person can never pronounce the same word twice exactly the same way, a pattern will never be incorporated into a program twice in the same way. The names used in a design pattern description can be changed in the actual program, for example, or the dimensions of architectural features altered to suit the building being built.

The other main parts of a pattern, the problem, context, discussion of forces and so on, are much more abstract than the concrete solution. The problem and context are tightly interrelated in that they present a qualitative analysis of the solution, and should comprise a convincing argument that the solution proposed by the pattern does in fact resolve the problem. The problem statement is typically a brief and pithy statement of the problem the pattern sets out to solve, while the context can be an extended description of a general area or kind of design, and may enumerate important issues (forces) to be resolved, or discuss why obvious or naive candidate solutions would not solve the problem satisfactorily.

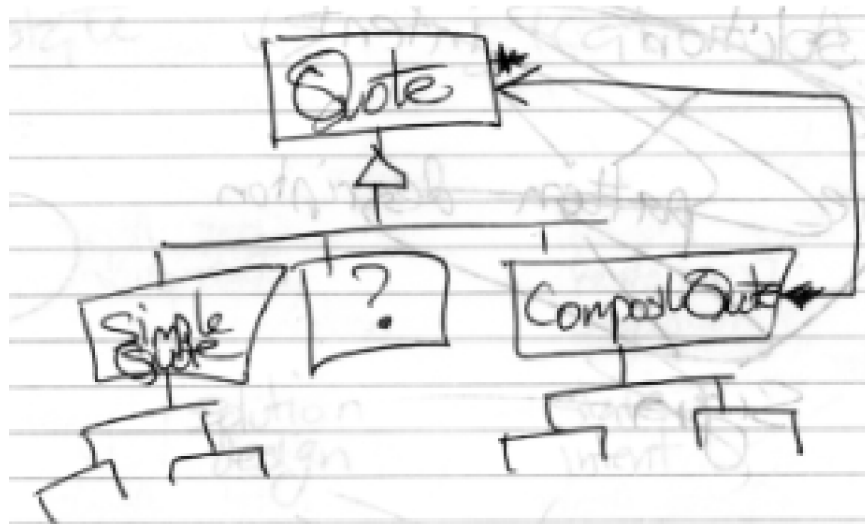


Figure 2: A sketch of a class diagram for part of a sales quotation system.

When reading a program or wandering around a building, we can see the concrete features of patterns: however, we understand those patterns as being more than just

their concrete features. For example, when we see a door under a set of stairs on the ground floor, we don't just think "*Oh, there's a door under the stairs on the ground floor*". Rather, we think "*Oh, there's a **Cupboard Under The Stairs***" — where CUPBOARD UNDER THE STAIRS (Rowling, 1997) is one of Alexander's patterns and we have recognised a particular token of that general type.

In the same way, when a programmer sees a class diagram sketched in a notebook or on a whiteboard (such as Fig. 2) or when they read a program's source code, they can see only the concrete structure — an inheritance hierarchy where a subclass has a one-to-many relationship back to its own superclass.

If the programmer understands patterns well, they could recognise Fig. 2 an application of the Composite pattern, and thus bring their knowledge of that pattern to bear without having to work it out from first principles — so, for example, they will immediately appreciate that:

- The program implements a recursive tree structure of Quote objects.
- A single quote or tree of quotes can be accessed uniformly via the common Quote interface.
- Whenever client code expects a Quote object, a CompositeQuote can be supplied instead.
- Client code is simplified, as it doesn't need to know whether it is dealing with primitive or composite Quotes.
- New kinds of Quotes can be added easily.
- Leaf nodes in the recursive composite all inherit from the SimpleQuote class
- Similar designs have been used in Interviews, ET++, Smalltalk, and many other systems since *Design Patterns* (Gamma et al., 1994).

Both of these examples, recognising the cupboard under the stairs and recognising the Composite pattern, involve signs. In each case, we see concrete features (signifiers such as doors, handles, classes, relationships) and then imagine abstract concepts (signifieds such as the Cupboard Under The Stairs and the Composite pattern) to make sense of those concrete features.

This, then, leads us to the first key point of this article:

A pattern-description is a sign, where the signifier is the pattern's solution and the signified is the pattern's intent, that is, its problem, context, known uses, and rationale .

The structure of this sign is illustrated in Fig. 3.

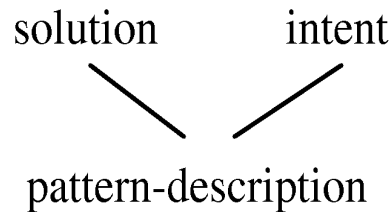


Figure 3: A pattern description as a sign.

Reading a program (or “reading” a building) is an example of semiosis, of sign exchange, in this case, producing meaning by exchanging the concrete signifiers for the abstract signifieds. Writing a program using patterns is also a process of sign exchange, producing a text of signifiers which are the concrete parts of the signs whose signifieds capture the meanings we need to embed into the program. Technically, patterns describe general *types* of problems and solutions; in reading or writing patterns we apprehend particular *tokens* of these *types*.

Patterns are not the *only* signs in a program: the lexical elements of a programming language can be considered as signs, as can algorithms, data structures, idioms, programming styles, and so on. We cannot construct the meaning of a whole program by considering each sign in isolation (as we have been doing here for the sake of a simple presentation): the meaning of a program (or a building or a novel or a movie) is produced by the combination of a large range of signs, where any particular sign’s meaning can be influenced and altered by its context, and by other signs in the text — for a simple example the signifier “!” in a Boolean expression in a C++ program forms a sign that negates signs in its subexpression.

4.2 A Discourse of Patterns

Representing pieces of knowledge about programming is not the only function of patterns. Patterns exist within a social context, where they provide a shared work-language with a common vocabulary that programmers can use to talk about design (Lea, 1994, Gamma et al., 1994, Coplien, 1996, Meszaros and Doble, 1998). Were a team of programmers working on the quotation system shown in Fig. 2, they would not just talk about the advantages of the pattern-based design versus other alternatives. Rather, every pattern has a name that programmers can use to refer to it: by saying “we could use Composite here”, for example, one programmer can communicate all the essential details of the design in Fig. 2 — both the basic shape of the final implementation and the underlying abstract intent, rationale, design tradeoffs that are part of the pattern.

This is another instance of semiosis — a word in a language signifying an abstract concept. In this case, the language is the human language spoken by the programmers, and the abstract concept is the pattern, that is, both the abstract concept of the pattern and a description of the concrete implementation. A pattern name is the signifier of a second sign of which the pattern-description is the signified.

This gives the second key point of this paper:

A pattern is a sign, where the signifier is the pattern's name and the signified is the pattern-description.

The resulting second-order semiotic system is shown in Fig. 4. This is a second-order system because it is composed of two signs, where one sign becomes a component of a second sign. It is a denotative system because the second-order sign directly names the first-order sign — that is, the first-order sign becomes the signified of the second-order sign. Technically, the pattern-names form a metalanguage of discourse about the base-language of pattern-descriptions (Cobley, 2001). Note that this is in contrast to many other semiotic systems, where the signified of a (denotative) first-order sign is the signifier of a (connotative) second-order sign: the second-order sign produces myths implicit in the first-order sign (Barthes, 1972). Here the first-order sign is the signified, so the second-order sign is denotative; we briefly address connotative (third-order) signification in patterns in section 8.5.

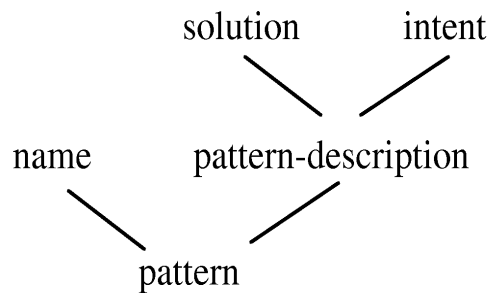


Figure 4: A pattern as a second-order sign

The second-order sign can also be “read” to produce meaning — when a pattern name is written or spoken, a reader or listener can construct the pattern-description as the meaning of that signifier; similarly, a pattern latent in a program can be named by the signifier. This is a second order process because, say, reading a program for patterns involves two stages of semiosis: first, the concrete implementation is exchanged for the pattern’s abstract intent, and second, this sign as a whole is exchanged for the name of the whole pattern. Similarly, hearing a pattern name as part of a conversation also invokes a two-stage process to construct its meaning: first, the pattern name must be exchanged for the first (pattern-description) sign, then the signified of that sign can be exchanged for the intent.

5 Questions about Patterns

There are a number of quite basic open questions regarding design patterns. Some of these questions are posed by novices to patterns, perhaps during their first reading of *Design Patterns*: other questions are more subtle, and arise only after more considered study, or experience attempting to write patterns.

In this section we show how a number of these questions can be addressed using our semiotic approach — beginning with questions of pattern descriptions, then pattern names, and finally considering the relationships between patterns.

In the spirit of the patterns movement, our proposed *answers* to these questions are not necessarily novel, rather, the contribution of this article is in the semiotic explanation of the answers to these questions.

5.1 Questions of Pattern Descriptions

We begin by considering questions relating to patterns' intents and designs — that is, questions of pattern-descriptions.

5.1.1 How can two patterns have the same implementation?

One common question asked about patterns is “What’s the difference between the Strategy and State patterns?” Both these patterns have almost identical structure diagrams, that is, solutions — Fig. 5 shows the two structure diagrams taken directly from *Design Patterns*. The names used for the components of patterns should be changed to fit a pattern’s context when it is applied: either pattern applied in the same context could produce the same solution. How, then, can they be different patterns? Would we not be better off with a single pattern encompassing both State and Strategy? (Agerbo and Cornils, 1998).

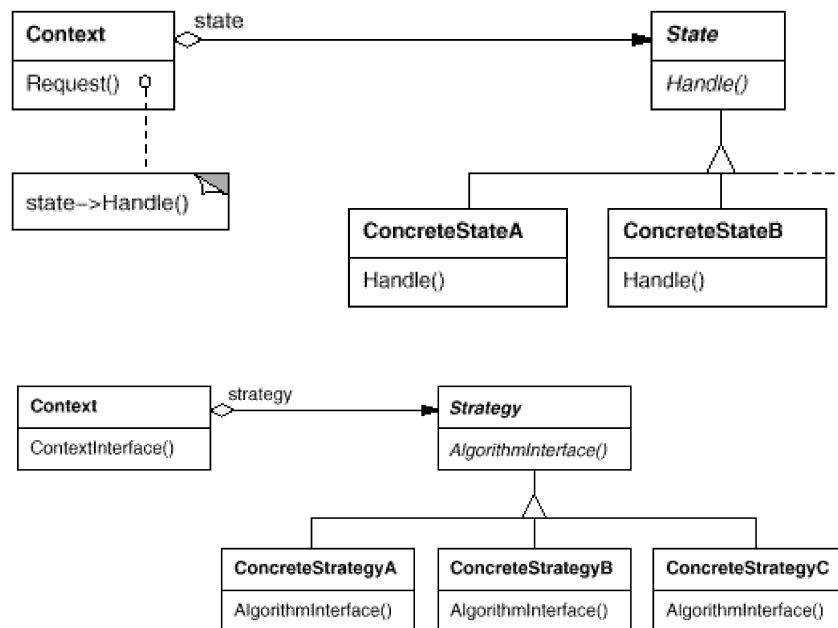


Figure 5: *Design Patterns* State and Strategy pattern structure diagrams (Gamma et al., 1994).

In terms of our semiotic approach to patterns, we can see this question as symptomatic of a misunderstanding about the nature of patterns: confusing signifiers and

signs. One signifier can form more than one sign, just as the English pronunciation “red” can signify both the colour red and the past tense of the verb “to read”. In the same way, a pattern description is a sign, not just a signifier, so the same signifier (the same implementation) can form part of more than one pattern description. In other words, a pattern is a solution to a problem, not just a solution. Fig. 6 shows the semiotic structure of these two patterns, each sharing a solution but with different intents and names.

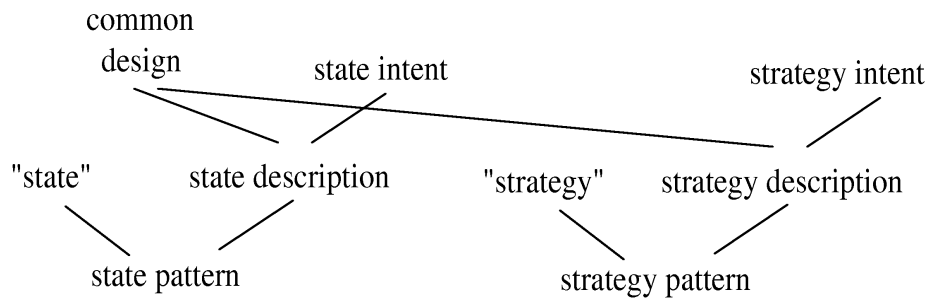


Figure 6: State and Strategy Patterns

5.1.2 How can one pattern have more than one implementation?

Sometimes the text of a pattern describes more than one implementation. For example, *Design Patterns* describes four separate variants of the Adaptor pattern — Class Adaptors that use (multiple) inheritance, Object Adaptors that use delegation, Two-Way Adaptors that again use multiple inheritance, and Pluggable Adaptors where adaption is built in to the adaptee classes. Each of these implementations have different advantages and disadvantages that are discussed in the consequences and implementation sections of the pattern.

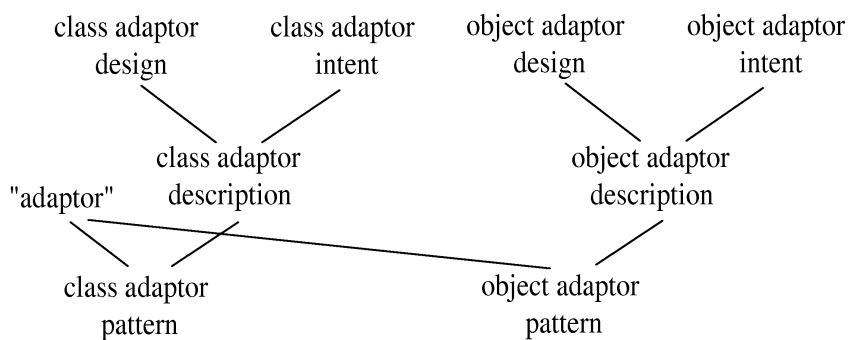


Figure 7: Polysemy in the Adaptor pattern

In terms of our semiotic model, we can see that each of these variants is effectively a different pattern-description (the first-order sign) — a different abstract concept (signified) with different consequences and tradeoffs, and obviously with a different de-

sign (signifier) — with, presumably, the same name at the second-order sign. This is not a problem *per se*, as multiple signs with the same signifiers are common in sign systems: a sign is not just a signified, but a relationship between signified and signifier. Technically, a signifier forming multiple signs is called *polysemy* (Cobley, 2001); the semiotic approach at least lets us analyse this cleanly (see Fig. 7).

In terms of the language used to communicate about patterns this causes certain practical difficulties: each of these different designs leads to a different sign (a different pattern) with the same name. These names can be disambiguated as necessary by other components of the message of which the polysemic signifier forms part, or by negotiation (Eco, 1997) — one developer may ask another: “Do you mean a Class Adaptor or an Object Adaptor?” A closer analysis shows that the text *Design Patterns* does this in practice, explicitly introducing extra disambiguating signs as we have done in this discussion. *Design Patterns* introduces particular names for the more radical variants: in the text, the phrases “pluggable adaptor” and “two-way adaptor” are printed in boldface, which is a sign that these phrases are important.

This gives an alternative interpretation in our model, where each adaptor variant is again a separate sign, but where the second-order signs differ not only in their signified but also their signifiers. In conventional pattern terminology, this can be expressed as each variant design giving rise to a separate “first class” pattern, each with its own name: Class Adaptor, Object Adaptor, Two-Way Adaptor, Pluggable Adaptor (Fig. 8 shows the first two patterns).

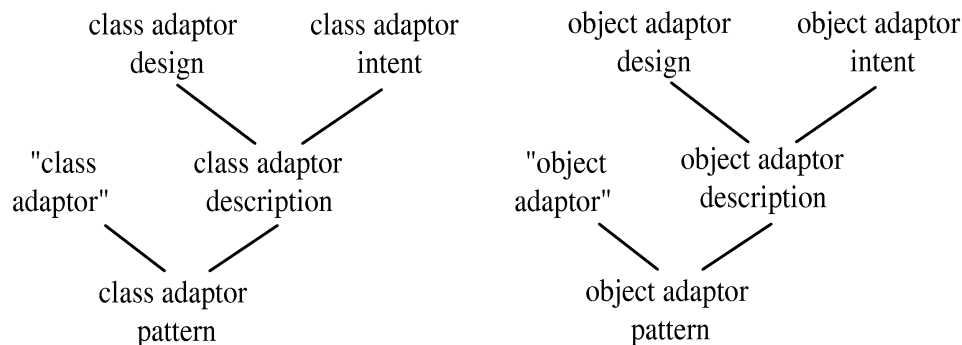


Figure 8: Disambiguated Adaptor patterns

5.1.3 How can one pattern solve two or more problems?

Complementing those patterns that have multiple solutions, some patterns are described as solving multiple problems with a single design. The best example here is the Proxy pattern: in *Design Patterns*, Proxy is presented as solving four different problems (protection, loading on-demand, remote access, pointer dereference), while *Pattern-Oriented Software Architecture* describes seven different problems that can be solved by the proxy pattern.

In terms of our semiotic model, this means that each pattern-description will be a different (first order) sign with the same signifier (the same design) but different signified, because the purpose of the pattern is part of its signified. In terms of the second order sign, often all these patterns have the same name (Fig. 9).

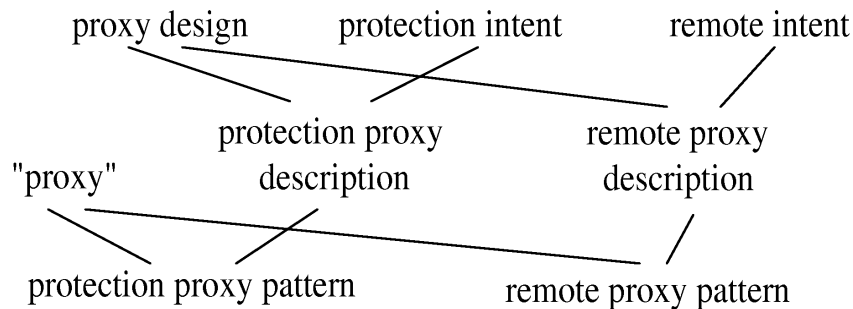


Figure 9: Multipurpose patterns

This is similar to the situation described above where one pattern has multiple designs, except here the fundamental difference between each pattern is in the intent (first order signified), rather than the design (first order signifier): the pattern name (second order signifier) is again polysemic. If each separate problem is in fact a separate sign, then each separate problem gives rise to a separate pattern: this would certainly follow naively from a pattern being defined as “*a solution to a problem in a context*”: here, although the solutions (and names) may be the same, the problems are certainly different.

Again, both *Design Patterns* and *Pattern-Oriented Software Architecture* tend towards resolving the ambiguity of the pattern names by introducing more specialised names for each particular problem. Thus there are Protection Proxies, Virtual Proxies, Remote Proxies, and so on, where each different pattern has a different name.

5.2 Questions of Pattern Names

As well as questions primarily related to pattern descriptions, there are also a number of questions relating to pattern names.

5.2.1 How can one pattern have more than one name?

Every pattern form ensures that each pattern has a name. Most large-scale pattern forms, however, allow a number of alternative names — synonyms for each pattern. In terms of the semiotic model, we must treat each as a separate second-order sign because the signifiers (names) are different, even though the first-order signs are the same.

Fig. 10 illustrates this for Decorator and its synonym Wrapper. What is interesting here is the way that names evolve to reflect different shades of meaning: treating each name as producing a separate (second-order) sign allows us to consider this evolution explicitly. For example, part of what it has meant for the *Design Patterns* book to become widely accepted is that the pattern names it proposes have themselves become the canonical names for the pattern-descriptions in the book, and almost all of the alternative names (even those proposed in *Design Patterns*) have fallen out of use. So, for example the alternative name “Kit” for “Abstract Factory” is no longer used; “Bridge” has replaced “Handle/Body” (although “Handle/Body” is arguably a more

descriptive name for the pattern); “Factory Method” has replaced “Virtual Constructor”; “Iterator” has replaced “Cursor”, and so on.

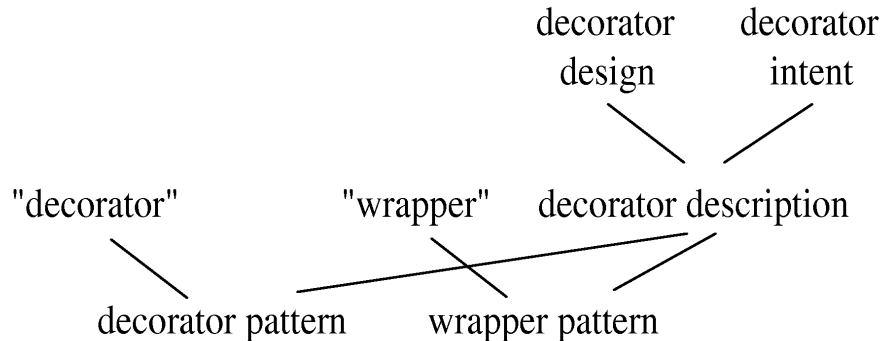


Figure 10: Patterns with synonyms

One case where this has not happened has been with the name “Wrapper”. *Design Patterns* gives both the Adaptor and Decorator patterns the synonym “Wrapper”; however the name “Wrapper” is still in general use both for Adaptors, Decorators, and also for Proxies. All these patterns are quite closely related; in particular, their implementations can be identical in many cases. The pattern-name Wrapper may be acting as a signifier for a more basic pattern describing the solution, where the intent is simply to “wrap” another object for whatever reason, and the other patterns — Adaptor, Decorator, and Proxy — could be (special kinds of) Wrappers used to solve more specific problems.

5.2.2 How can many different pattern-descriptions share the same name?

The complementary problem to one pattern having many different names is where one name is used for many different patterns². For example, in the *Patterns Almanac* (Rising, 1999) there are a number of patterns with the name “Prototype” — the Prototype pattern from *Design Patterns*; Prototype from Coplien’s *Generative Development-Process Pattern Language* (Coplien, 1994); a similar pattern from Cockburn’s *Surviving Object-Oriented Projects* (Cockburn, 1998): the almanac also lists at least three other patterns named as some variation on “Prototype”, and no doubt more have been published subsequently. Similarly, a recent J2EE textbook (Crupi et al., 2001) includes a pattern named “Value Object” which is quite different from existing patterns called “Value Object” (Fowler, 2001, Kühne, 1999).

In the patterns community, control of pattern names is an important issue, and a significant part of a crucial problem: how to index and identify patterns. The *Patterns Almanac* (Rising, 1999) is the most successful attempt at building such an index so far, but it suffers from many duplicate named patterns. Some of these duplicates may be almost unrelated (“Prototype-Based Object System” and “Prototype And Reality”) while others may be very closely related, as in Coplien’s and Cockburn’s respective Prototype patterns. Furthermore, a single “pattern” can be described in a number of

² This problem was identified by Linda Rising.

different versions — very similar Proxy patterns have been published in both *Design Patterns* and *Pattern-Oriented Software Architecture*.

In terms of the semiotic approach, we can see this as each pattern being a separate sign; however, the two second order signs each share the same signifier (the same structure as Fig. 7) — in much the same way the spoken English word “red” can form part of two signs. In general conversation, we can distinguish the intended pattern according to context — disambiguating via other signifiers in the message containing the term “prototype” and explicit bibliographic references if necessary. Rather than attempting to privilege one final “best” description, the semiotic approach can facilitate negotiation and discussion, highlighting relationships and differences between several patterns.

6 Relationships Between Patterns

Semiotics, being fundamentally concerned with the differences and relationships between signifiers, signifieds, and signs, can also help define the relationships between patterns (Zimmer, 1994, Meszaros and Doble, 1998, Noble, 1998). Some pairs of patterns will fundamentally be different: that is, both their signifier (design) and signified (intent) will be mutually unrelated. More interesting cases arise when one (or both) of the parts of a pattern are *similar*, yet the pattern-descriptions as a whole differ.

6.1.1 Uses

The primary relationship between patterns is that one pattern may use another pattern in its implementation. “Uses” is also known by longer names, including “requires”, “completes”, or “follows”, (although “follows” can also mean that one pattern is printed after another pattern in an Alexandrian pattern language). The key to this relationship is that one pattern must be applied as part of applying the other pattern — for this reason, the larger pattern is often called a *compound* pattern (Riehle, 1997a, Vlissides, 1998).

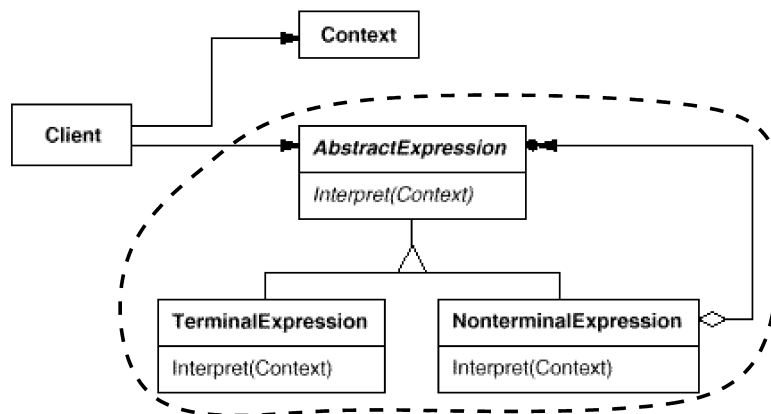


Figure 11: *Design Patterns* Interpreter structure showing Composite substructure (Gamma et al., 1994)

The classic example from *Design Patterns* is the relationship between Composite and Interpreter: as part of applying the Interpreter pattern, you must apply the Composite pattern to represent the language being interpreted (Fig. 11).

In our semiotic approach, we recognise this relationship where two patterns have a different intent (Composite models recursive structures, Interpreter interprets a language), but where the implementation of the larger pattern is related to the pattern it uses, as Interpreter’s implementation is related to the whole Composite pattern; see Fig. 12.

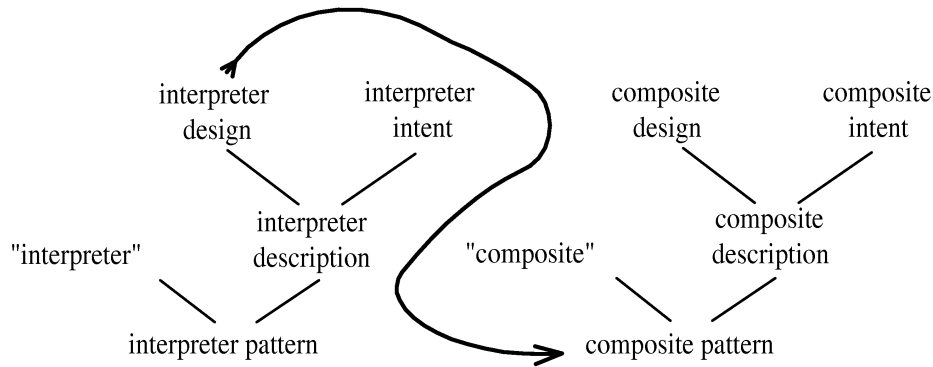


Figure 12: Composite and Interpreter Patterns

6.1.2 Alternative

A second relationship between patterns is that two patterns can be alternatives, that is, they provide different implementations to address (some of) the same problems. *Design Patterns*, for example, discusses how Decorator and Strategy provide alternative designs to address problems of adding and changing responsibilities of objects, possibly dynamically. A Decorator changes the “skin” of an object, changing it from the outside by adding a transparent wrapper, while a Strategy changes the “guts” of an object, possibly requiring the object to be changed to be aware of the extension (Gamma et al., 1994, p. 180). Both Strategy and Decorator are applicable to a wide range of common problems, such as adding graphical decorations (title bars, close buttons) to windows, or adjusting event-handling behaviour, but both clearly present different designs and have some different consequences.

In our semiotic approach, we recognise this relationship where two patterns have a similar intent (both Decorator and Strategy allow programmers to change objects), but where the designs that support these intents are different (Figure 13).

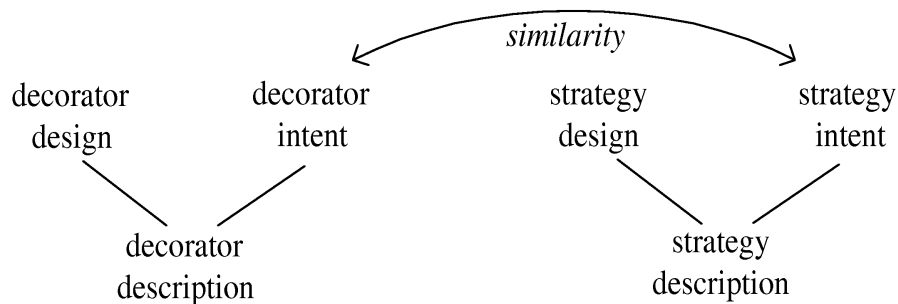


Figure 13: Decorator and Strategy Patterns

6.1.3 Specialisation

The third primary relationship between patterns is that one pattern can be a specialisation of another (conversely, the second pattern can be a generalisation of the first). *Design Patterns* again provides several examples, for example, a Factory Method is a special kind of Hook Method that creates objects. In our semiotic model of patterns, we recognise this relationship when two patterns present similar intents and similar designs, but the more specialised pattern is more complex than the more general pattern (see Fig. 14).

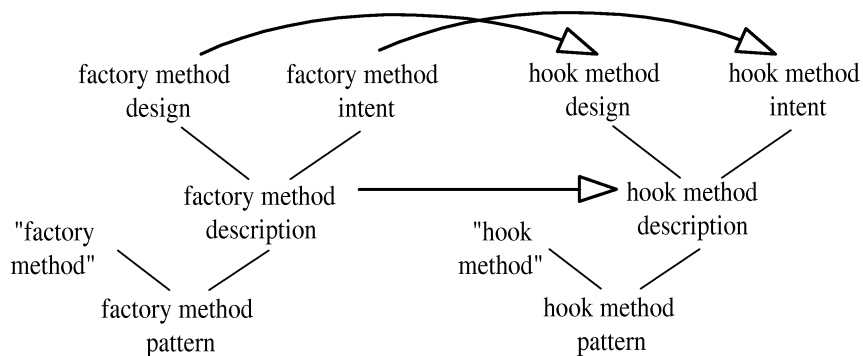


Figure 14: Factory and Hook Methods

For example, considering intent, both Factory Method and Hook Method allow subclasses to modify behaviour defined in their superclasses (similar intent), however Factory Methods modify this behaviour to change the type of object created (changing a particular kind of behaviour). Considering implementation, both Factory Method and Hook Method are typically implemented by specially-named abstract (C++ pure virtual) methods that must be redefined in subclasses, however a Factory Method must return an object which is in some sense “new”, whereas the behaviour of a Hook Method (qua Hook Method) is undefined.

Figure 14 shows how specialisation occurs primarily between first-order signs. Especially after disambiguating variants, the names of a specialised pattern may be

related to a more general pattern (a “Protection Proxy” is a special kind of “Proxy”) so there may also be a specialisation relationship in the second-order sign.

7 Patterns, Reference, and the Rhetoric of Object-Oriented Design

So far in this article, we have treated patterns and their relationships separately from other parts of the design of an object-oriented program. In this section, we will extend our semiotic model to illustrate how the kinds of designs generated by patterns differ from more basic object-oriented designs. To do so, we need to consider both signs within programs and their relationships with the parts of the real world that the program represents. For this analysis, then, we follow Eco (Eco, 1976) and adopt Peirce’s triadic model of the sign (Peirce, 1934-1948) shown in Fig. 15.

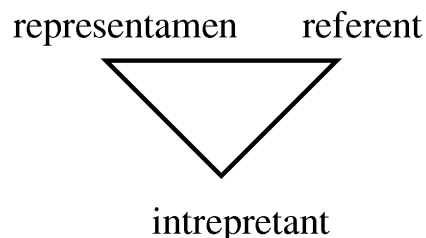


Figure 15: Peirce’s Sign

Peirce’s sign is a three-part relationship between a *representamen*, a *referent* and a *intepretant*. The representamen (similar to Saussure’s signifier) is some phenomenon that an individual can see, hear, sense, or imagine. The referent (Peirce’s term is *object*: we shall use referent in this article to avoid confusion between semiotic objects and the objects in object-oriented programs) is the concept or entity to which the referent refers: the “something else” for which the referent stands. Finally, the interpretant (closest to Saussure’s signified) is the mental concept that the representamen produces.

For example, consider the English word “chocolate” as a Peircian sign. The spoken or written word “chocolate” is the representamen; a solid compound of cocoa beans, cocoa butter, sugar and milk is the referent; the resulting mental concept of chocolate in the reader or hearer of the word is the interpretant.

Peirce also developed a complex and intricate typology of signs — luckily, we will need only the most basic classifications in this article. *Iconic Signs* represent their referent by a direct likeness, as picture of a chocolate bar. *Indexical Signs* represent their referent by some connection such as an attribute, cause, effect, or part of a whole, such as a Swiss mountain or a cow used to advertise chocolate. Finally, *Symbolic Signs*, or *symbols* have a purely arbitrary, conventional connection to their referent, such as word “chocolate”.

7.1 Object-Oriented Design

A program execution is regarded as a physical model, simulating the behaviour of either a real or imaginary part of the world.

Object-Oriented Programming in the BETA Programming Language. Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard (Lehrmann Madsen et al., 1993)

This quote from Lehrmann Madsen et al. outlines the core principle underlying object-oriented design, that an object-oriented program simulates (or models) the world. To model a farm, for example, a program could have a Bovine class, where each object represented a cow; an Ovine class where each object represented a sheep; a Porcine class where each object represented a pig, and so on (Noble et al., 2002). Following a Peircian approach we can take this simulation or modelling relationship to be semiotic: that is, we can treat a program as a sign where the representamen is the set of objects and classes in the program; the referent the set of entities in the world that the program models, and the interpretant the concept that a particular program models a particular real or imaginary subpart of the world (Andersen, 2003). For an object-oriented program, we go further, taking a particular object in the program as a representamen; the part of the world that object models as a referent; and the idea that a particular object models a particular piece of the world (that the Bovine instance at memory location 0xDEADBEEF models Daisy the cow in the field somewhere) as the interpretant of the sign. (see figure 16).

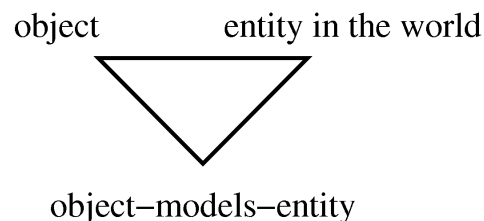


Figure 16: An Object as a Sign

As the focus of this paper is primarily on patterns, rather than objects, we will not elaborate further on the semiotic structure of object-oriented programming here: this is an area of active research (Andersen, 1997, Andersen, 2003, Noble et al., 2002).

What we wish to address in this section is the *type* of sign produced first by classical object-oriented design, and second by design patterns. All these signs are denotative, as the elements of the program refer directly to the elements in the world that they model. Technically, all these signs must be conventional *symbols*, because there is no real likeness or direct causal relationship between the representamen and the referent: i.e. between the text of the program and the world it models (Andersen, 1997). At the most basic level, every sign in the execution of a computer program is symbolic: all representamens are patterns of electrons in memory cells under an arbitrary binary encoding. Those computer systems that are causally connected to an external world may contain indexical signs. This connection may be direct, as in a computer-controlled automotive brake where objects in the program will be casually connected to the state of the driver's brake pedal and others to the brake actuators on each wheel, or indirect, as in a banking system where tellers access and update account holders' details via a screen and keyboard. A few computer-based signs —

when rendered by appropriate output devices — are iconic: a digital portrait displayed on a high-resolution screen is as much an icon as an oil-painting or photograph.

In this section, however, we wish address the way the rhetoric of object-oriented design represents the relationship between objects in programs and the world they model, and the way that relationship changes when designs are produced by patterns. Simply saying that all computational signs are symbolic, depending on a binary code, is fine as far as that takes us: but it doesn't take us very far. Andersen has described object-oriented modelling in terms of *reification* — creating objects to model design concerns (Andersen, 2005). Reification is clearly at the core of object-oriented design, but here we hope to make a finer distinction between types of objects, or types of reifications, that are constructed in designs.

The core problem we wish to address is that many texts on object-oriented design ranging from books aimed at novices (Korienek and Wrensch, 1991), practitioners (Rumbaugh et al., 1991, Henderson-Sellers, 1994, Henderson-Sellers and Edwards, 1994, Lehmann Madsen et al., 1993) to theorists such as Abadi and Cardelli (Abadi and Cardelli, 1996) treat objects in programs as signs that are *not* arbitrary: rather, they are motivated by the structure of the part of the world they have been designed to model. Responsibility Driven Design, for example, advises developers to model physical objects and conceptual entities but *not* attributes of objects (Wirfs-Brock et al., 1990). These answers have often become more sophisticated over time. In the first edition (1988) of *Object Oriented Software Construction* Meyer gives the (admittedly premature) advice that “objects are just there for the picking”; the second edition is rather more circumspect (Meyer, 1988, Meyer, 1997). General practice within object-oriented design, however, is overwhelmingly of the opinion that the objects in the program should be chosen to correspond to their real-world counterparts as closely as possible (at least with respect to those parts of the real-world with which the program is concerned).

From this perspective we make a preliminary claim that the rhetoric of object-oriented design methodologies aims to establish *iconic* relationships between the objects in a program and their referents: at some level, the difference between a picture of a pig (clearly an iconic sign) and a Porcine instance in a program is one of medium, not of typology. Following Peirce (Bergman and Paavola, 2005, Hiraga, 1994), we can consider object-oriented models as *diagrams*, that is as *structural* icons: the structure of an object-oriented model forms a likeness of the structure of the parts of the real world it is intended to model. We claim this structural iconicity applies to written or printed diagrams of the program, as well as to the binary or internal representation of the program's execution.

7.2 Iconic Patterns

Some patterns produce designs that are structurally iconic in this sense. There appear to be several kinds or structures of objects that are common in the worlds that programmers wish to model, but which cannot be translated directly into object-oriented designs — perhaps due to weaknesses or non-orthogonality in programming or modelling languages (Coplien and Zhao, 2000, Gabriel, 1996).

For example, object-oriented modelling languages such as UML support multi-directional, multi-place relationships between objects (Fowler and Scott, 1997). Such relationships can be used to model the relationships and collaborations between entities in the real world. In a university course database, for example, the relationship

between students and courses is a many-to-many bidirectional relationship: multiple students can study multiple courses, students need lists of the courses they are studying, and lecturers need lists of all students enrolled in their courses. Few (if any) object-oriented programming languages support bidirectional multiway relationships, so this relationship needs to be encoded somehow into the program, typically by introducing an extra “relationship object” to reify the relationship (Noble, 2000).

Design Patterns includes several patterns that produce these kinds of models. To give two examples, the Singleton pattern is a simple code idiom that ensures that a particular class may have only a single instance; and the Adaptor pattern describes how an extra object can be used to change an object’s interface, in the same way that a travel adaptor can enable a European mains plug to draw power from an American socket.

7.3 Indexical Patterns

The designs produced by many other design patterns don’t make sense considered as this kind of iconic pattern, however. Consider again the State pattern, one of the simpler patterns (see Fig. 17). The state pattern allows an object (the Context) to alter its behaviour when its internal state changes, causing the Context object to appear to change its class. As the figure shows, the State pattern introduces a state object aggregated inside the context, and delegates some requests to it. The internal state object is an instance of a ConcreteState class (where the ConcreteState classes all inherit from a common abstract State class). The behaviour the context object receives when delegating requests to the state object will change according to the ConcreteState object that is installed at any time, so by changing state objects dynamically the whole context object can provide different behaviour. The State pattern is typically used to manage input modes (reflecting the global state of a user interface) and communication protocols (reflecting the state of a network connection). In a farm system, for example, we could use the State pattern to record the changing state of a sheep from newborn lamb, drenching, crutching, dagging, hogget, breeding ram, and finally to mutton.

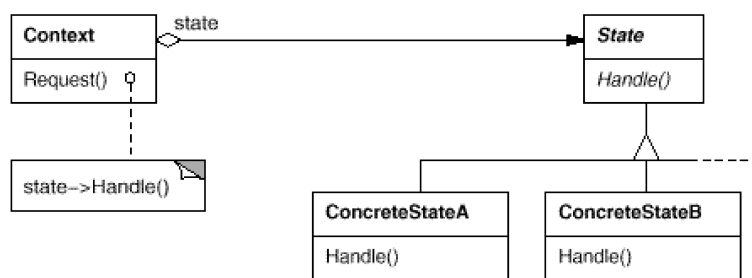


Figure 17: The structure of the State pattern (Gamma et al., 1994)

Now, the implementation of the State pattern is quite straightforward: a developer must add an extra object and class hierarchy to their design, and then change internal state objects to change context objects’ behaviour. The State pattern does not depend on programming language features: while some languages can reduce the amount of

code required, the overall geometry of the solution remains the same (Ungar and Smith, 1991, Taivalsaari, 1993, Ernst et al., 1998).

On the other hand, the design produced by this pattern is not structurally iconic as we described object-oriented design in section 7.1 above. In the farm example, the sheep state object does not model a subordinate physical object that is attached to a sheep and that changes throughout the sheep's lifecycle. In fact, there may be no physical change at all between a sheep considered a newborn lamb one day, a yearling the next, and a prime export candidate the day after.

Consider the structure of another common pattern, the Iterator pattern (see Figure 18). The Iterator pattern is used to traverse through a collection object such as a list: the Iterator acts as a traversal cursor, keeping track of a position in the list, and is able to provide the `CurrentItem` at that position and advance to the `Next` item in the collection. Iterators are very straightforward to implement, and form part of the design of the fundamental library for most object-oriented languages, including Smalltalk, C++, and Java. Again, these Iterator objects do not form iconic signs of objects in the world of the program. Furthermore, unlike a State object, an Iterator must be part of the collection's public interface, and cannot be dismissed as a mere implementation detail.

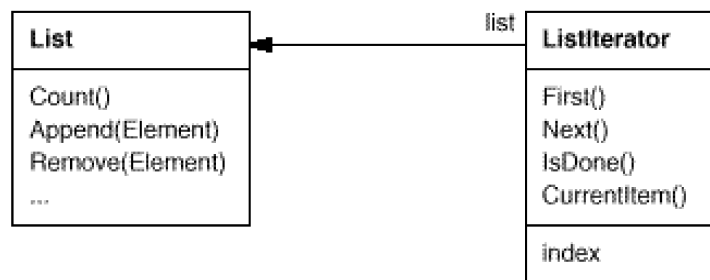


Figure 18: The structure of the Iterator pattern (Gamma et al., 1994)

For these reasons, we consider that State and Iterator objects, and the similar objects introduced by many other patterns, are exemplars of *indexical* designs, rather than iconic designs. They do *not* structurally represent objects from the real world, rather they model some attribute, cause or effect, typically of some other real-world object. The states of the sheep, for example, are not icons for “real” objects own right, rather, they signify attributes of their referent. Similarly, the position of an iteration is not part of reality, rather it is a consequent effect of iterating through the collection.

The majority of the design patterns produce indexical designs, rather than iconic designs. The Abstract Factory, Builder, and Factory Method patterns, for example, are about causing other objects to be created; while the Command, Decorator, Strategy, Mediator, Memento, and Visitor patterns model activities, responsibilities, algorithms, collaborations, snapshots of internal states, and traversal operations, respectively (Gamma et al., 1994).

Perhaps the most commonly used design pattern, the Composite pattern (also described as Part-Whole in *Pattern-Oriented Software Architecture*, and discussed in section 4.1 above) is also indexical. Composite allows programmers to “treat individ-

ual objects and compositions of objects uniformly”, that is to take a part for a whole and vice versa. This relationship between representamen and referent is indexical: just as a picture of a part of a person, such as their passport photograph or fingerprint, can stand for a whole person.

7.4 Symbolic Patterns

Finally, there are some patterns that are neither iconic nor indexical. Consider the Facade pattern, for example (see Figure 19). The Facade pattern inserts an extra interface into a program to encapsulate a set of objects forming a subsystem. A Facade will have a structural relationship to the objects it encapsulates, but its semiotic relationship is much more tenuous than a pattern like State: necessity has caused the programmer to introduce another representamen; how it stands to a referent is unclear. Flyweight is another pattern in this category. The intent of the Flyweight pattern is to support large numbers of fine-grained objects efficiently: it achieves this goal by sharing some (so-called extrinsic) attributes between many objects.

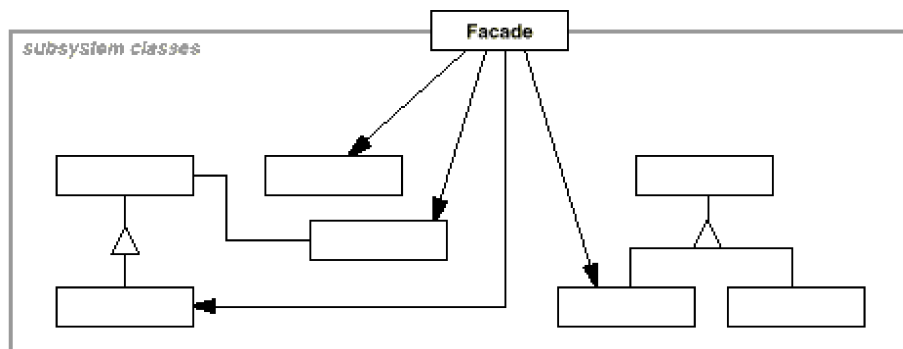


Figure 19: The Facade pattern (Gamma et al., 1994)

We consider that the objects produced by these kinds of patterns participate as purely as *symbolic* signs: they have only a conventional relationship with the world the program has been constructed to “simulate”. Many of the patterns in the book *Pattern-Oriented Software Architecture* are symbolic patterns, including the Layers, Blackboard, and Pipes and Filters patterns.

(As an aside, note that however designs or patterns actually function, the *names* of patterns are almost always *metaphorical*. For example, the name of the Bridge pattern acts as a metaphor for the *class diagram* of the structure of the pattern — the diagram (see Figure 20) is supposed to look like a bridge between the “Abstraction” on the left and the “Implementation” on the right. In spite of this name, the bridge pattern, focusing on separating abstractions and implementations, is symbolic in that it has no structural relationship to the world being modelled.)

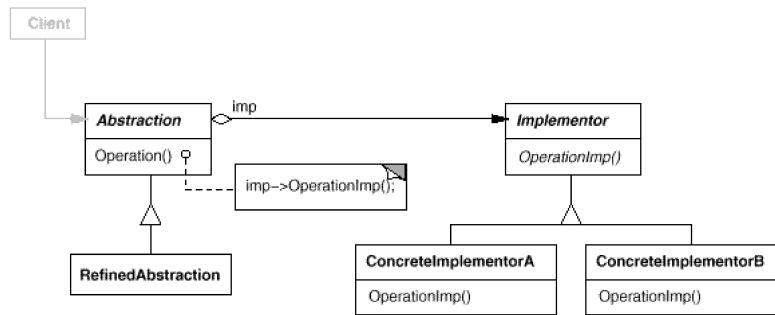


Figure 20: The structure of the Bridge pattern (Gamma et al., 1994)

The final pattern we will consider in this section is the Strategy pattern. We have presented the structure of this pattern (along with the State pattern) in Fig. 5: a context object delegates requests to an internal, replaceable strategy object that implements an algorithm, so changing the strategy object changes the algorithm executed by the context.

We consider both these patterns to be indexical: the strategy object reifies an effect of its context. In some applications, however, the pattern can be used in other ways. *Design Patterns* describes how currency trading pricing algorithms can be implemented as strategy objects. In such an application, the algorithm is part of the domain, so the design is iconic. Furthermore the pattern can even be used symbolically, as Strategies can be used to avoid multiple inheritance, reducing the number of classes in the program. Ultimately, although patterns mostly produce indexical or symbolic designs (Table 1 shows our typology of designs most commonly produced by each pattern) a particular application of a pattern can fall into any category.

Table 1: Classification of Patterns

Creational Patterns		
Abstract Factory	Creates related objects	Indexical
Builder	Creates complex objects	Indexical
Factory Method	Creates subclasses	Indexical
Prototype	Exemplary object	Indexical
Singleton	Single Instance	Iconic
Structural Patterns		
Adaptor	Converts interfaces	Iconic
Bridge	Decouple abstraction and implementation	Symbolic
Composite	Model recursive tree structure	Indexical
Decorator	Add responsibilities to objects	Indexical
Façade	Interface for a subsystem	Symbolic
Flyweight	Save memory of similar objects	Symbolic

Proxy	Surrogate for access control	Indexical
Behaviour Patterns		
Chain of Responsibility	Handle requests	Indexical
Command	Request as an object	Indexical
Interpreter	Interpret a language	Symbolic
Iterator	Iteration cursor	Indexical
Mediator	Encapsulate interactions	Indexical
Memento	Snapshot of object's state	Indexical
Observer	Update dependents	Indexical
State	Change behaviour	Indexical
Strategy	Vary algorithms	Indexical
Template Method	Subclasses change algorithms	Symbolic
Visitor	Represent traversal operations	Indexical

This typology and classification raises the question of why most patterns produce indexical or symbolic designs, when we have argued that a key feature of object-oriented methodologies is that the designs they produce are iconic. Are design patterns somehow “less” object-oriented — or at least bad design practice — because the designs they generate are not iconic? We hypothesize, rather, that the emphasis on iconic design within object-oriented methodologies and academic or industrial training courses has meant that many object-oriented programmers will only have experience with iconic designs. Whether due to the structures in the referent world or the constraints of a programming language (and on the representamens that they can produce) certain situations will require indexical or symbolic modelling. Where these situations are common, they give rise to patterns³. These patterns then appear counter-intuitive to novice designers and programmers, because they are not used to the mode of reference that those patterns embody.

Recently, Peter Bøgh Andersen has proposed an alternative modelling ontology, centred around *activities* taken from the domain instead of *objects* (Andersen, 2005). In such an activity-oriented modelling system, activities and collaborations can be modelled directly — iconically, in our classification — where in an object-oriented model they have to be represented indirectly, generally via indexical patterns such as the Command or Mediator patterns. On other hand, it appears that activity-oriented modelling “may get into trouble with traditional inherent properties” of objects in the domain (Andersen, 2005) — inherent properties that would be modelled directly, by objects, in an object-oriented model. To address this problem, activity-oriented models can include specialised activities, called measurements that reify object's properties. In our classification, we consider it may be possible to treat measurement activi-

³ It is suggestive that the Composite pattern is one of the most common patterns, while synecdoche is one of the most common linguistic tropes. Both of these facilitate treating wholes as parts and vice versa.

ties as the application of a design patterns for activity-oriented design, in this case, producing indexical signs in our taxonomy.

8 Discussion

In this section we discuss further aspects of the semiotics of patterns and outline some future directions for this work.

8.1 Misinterpreting Patterns

One of the biggest challenges in documenting patterns is to avoid their misinterpretation⁴ — that is, that someone reading a description of a pattern will not understand (or understand imperfectly) the solution and the intent of the pattern being described. When reading a program (or inspecting a building), we can similarly misunderstand the patterns we find — the “cupboard” under the stairs is really a staircase to the basement office, the door we expect to push must be pulled, and the code we think is the Observer pattern is actually using Mediator, or is just a random bad design, and so on.

Our semiotic approach encompasses misinterpretation by making the possibility explicit. While signifiers, by their nature, are concrete, tangible, and therefore public, signifieds are abstract, intangible, results of private mental processes — when reading a program or exploring a building, each of us alone constructs signifieds of any signs we encounter. Due to this, it is perfectly possible to produce an “incorrect” mental image of a signified for which a given signifier stands — where “incorrect” here presumably means an signified that is somewhat different from that intended by the author of the sign.

For example, upon reading some code or seeing a messy sketch like Fig. 2, we could misinterpret the design as supporting the Decorator pattern (by missing the scribbled asterisk for the many-to-one relationship); the Proxy pattern (by a more general confusion); or even as the Prototype pattern (through ignorance, through weakness, or through our own deliberate fault).

Semiotics makes clear that these kinds of misunderstandings can happen whenever you use signs, so it should not be surprising that such misunderstandings arise with patterns. Eco (1976) describes semiotics as “the theory of the lie” precisely because these misunderstandings are possible: we may construct a different signified to that intended by the author of the signifier (especially when signifiers are polysemic); an incorrect signifier can be maliciously presented or chosen in error; we could accidentally interpret something as a signifier when it is merely decoration; and so on.

In practice, Eco argues, we negotiate to clarify communication, repeatedly exchanging our private concepts and eventually converging on an agreed shared public “meaning” (Eco, 1997) — “red” means red (or “observer” means Observer) because the speakers of the language tacitly agree on this sign. In programming language design, attention has been recently called to the need for secondary notation, such as comments, even in novel visual forms for programming (Petre et al., 1997). In the patterns community, the shepherding and workshopping of patterns at PLoP confer-

⁴ This observation is due to Frank Buschmann.

ences provides an explicit forum for these negotiations, and thus helps to manage misinterpretation of patterns (Coplien, 2000b).

8.2 Patterns and Pattern Languages

Alexander's architectural patterns are contained within a larger structure of patterns known as a *pattern language* (Alexander, 1977, Alexander, 1979) — a tree or directed graph of patterns, similar in structure to a formal grammar. Each individual pattern provides a single solution to a single problem, and then, like a production rule in a grammar, uses (*leads to* or *contains*) other patterns that address subproblems raised by that solution. The language begins with an *initial pattern* (like a grammar's start symbol) addressing a large scale problem — how to organise all of human habitation — of which all the other patterns transitively form subparts. The key advantage Alexander claims for this structure is that it guides the reader through the process of design: beginning at the initial pattern, *A Pattern Language* provides complete instructions from large scale town planning down to decorating the edges of window-sills (Alexander, 1977).

This is a fundamentally different structure from that used in “catalogues” or “systems” of patterns such as *Design Patterns* or *Pattern-Oriented Software Architecture*, which are primarily collections of individual patterns. This difference gives rise to questions such as “*How can a collection of patterns be transformed into an Alexander-style pattern language?*”⁵

Given the structural differences between a pattern collection and a pattern language, converting a collection into a language would require a major refactoring of the collection (Schmidt et al., 2000). To ensure one pattern relates one problem to one solution, we would need to “normalise” the patterns — ensuring each pattern describes a single solution to single problem, splitting problem variants (like the multiple uses of Proxy) and solution variants (like the multiple designs for Adaptor) into separate smaller patterns. Then, many more patterns would have to be written to meet the structural constraints of a pattern language: the *Design Patterns*, say, are nowhere near a complete prescription for producing whole programs, as there is no initial pattern (presumably describing how to build any kind of system) of which all the other patterns eventually form subparts.

The semiotic approach offers an alternative organisation for collections of patterns. While grammars are useful for describing which sentences are correct, they do not describe the semantics of those sentences: rather, dictionaries and encyclopædia describe the vocabulary of languages in terms of the semantics of signs and the relationships between them (Cobley and Jansz, 1997, Eco, 1976, Eco, 1997). Indeed, most pattern books are structured this way, to a greater or lesser extent (*Design Patterns* even describes itself as an “encyclopedia” (sic) (Gamma et al., 1994, p.357)). Compared with a pattern language, an encyclopædia admits a richer description of the relationships between patterns, with not just the uses relationship, but also alternative, specialisation, and arguably many secondary relationships as well (Noble, 1998).

An encyclopædia can be very similar to a pattern language in places. Where one compound pattern uses another pattern (as with Composite and Interpreter in section 6) a structure like a pattern language is created in a localised part of the pattern

⁵ This question was posted to the design patterns mailing list by Mark Ratjens on 2 July 2001.

collection, as and when it makes sense. Unlike a pattern language, this structure does not have to encompass the whole encyclopædia, so a pattern author is not required to provide an initial pattern describing a single large-scale problem, to ensure all patterns are subparts of the initial pattern, or to omit patterns that do not fit.

Our semiotic approach allows us to describe a common vocabulary of patterns that evolves over time, facilitated by negotiation involving its users, and so allowing an evolutionary, rather than prescriptive, form of progress. New patterns can be added to the vocabulary (or old patterns removed) without affecting its underlying structure, in the same way that entries can be added to or removed from an encyclopædia without affecting the integrity of the encyclopædia. Several later patterns (such as Null Object (Woolf, 1998), Value Object (Kühne, 1999, Bäumer et al., 1998, Fowler, 2001), and Role Object (Bäumer et al., 1999)) have effectively been added to the vocabulary originated by *Design Patterns* while some (such as Builder or Interpreter) have almost fallen out of common use.

In contrast, because of its tight structure, an Alexander-style pattern language is fundamentally closed: it is hard to incorporate new patterns into a pattern language because the pattern language's internal consistency — that there is one initial pattern to which all others are subordinate, and which addresses one single unified problem — must be preserved.

8.3 On the Naming of Patterns

The Naming of Patterns is a difficult matter — it isn't just one of your holiday games (Eliot, 1962, Meszaros and Doble, 1998, Coplien, 1996). As we have described above (5.2) many pattern names are polysemic, that is, ambiguous. In a controlled, structured, pattern language, the authors of the language are responsible for all the names of the patterns, and can choose unique names to avoid this problem. In a more open encyclopædia of patterns, naming patterns becomes more difficult, especially as authors can contribute individual patterns without prior coordination. For example, a recent Java patterns textbook (Crupi et al., 2001) includes a pattern named “Value Object” which is quite different from existing uses of “Value Object”, also captured in pattern form (Fowler, 2001, Kühne, 1999): this duplication has engendered some debate in the patterns community.

To avoid these kinds of problems, other scientific disciplines (Biology and Zoology in particular) have international covenants on naming — the International Conventions on Biological Nomenclature (ICBN) (Greuter et al., 2000) and Zoological Nomenclature (ICZN) (Stoll, 1964). These conventions establish overarching naming schemes, international governing bodies to manage them, and strict protocols controlling naming rights — for example, the proposed name of a new species may not be released publicly prior to its confirmation, which may be years after the species was identified. The names issued by the schemes are not arbitrary, rather they are chosen to indicate a position in a Linnaean hierarchy, and are also allocated to reflect the priority of the discoverer. To ensure names are unique and to avoid bias these names are generally given in Latin, such as “*Lycaeides sublivens Nabokov*”. As a result the names are not straightforward, and so are only used by professionals in the field: few people would call a grizzly bear an “*ursus horribilis*”.

It is unlikely that such a solution would be practical for managing the names of software patterns. Simply establishing a naming scheme is not sufficient: without a controlling body the scheme could not be enforced, and without that body exercising

a high level of crucial scrutiny, problems with name choices would simply reoccur within the scheme — in the same way problems arise within the Internet Domain Name scheme. For example, a rogue pattern author could choose the most popular names for their favourite patterns, thus denying them to other, more appropriate patterns. Furthermore, as most programmers do not speak Latin, it is most likely that patterns' common and ambiguous names would continue to be used much as they are now.

For these reasons, we expect the cost of such a scheme would outweigh its benefits. The semiotic perspective adopted in this paper accepts such name clashes as fundamentally unavoidable but generally unproblematic consequences of the open and evolving nature of an encyclopædia of patterns. In Saussure's semiotics, the *langue* is a set of shared rules and knowledge held by a language community — no member of the community may know the entire *langue* — and it will evolve to meet the needs of speakers to communicate precisely (Bauer, 1998). Two ambiguous pattern names can always be disambiguated explicitly in practice: e.g. “J2EE value object” versus “‘vanilla’ value object” (Raymond and Steele, 1993), and the name that is used most often will tend to displace other similar names from general usage in any given community.

8.4 Patterns of One Example

Patterns can embody a critical analysis of software or design practices, as, for example, Foote and Yoder's *Big Ball of Mud* demonstrates (Foote and Yoder, 2000). To ensure patterns describe designs that are worth repeating, the patterns community's value system requires more than one example for a pattern (Coplien, 1996, Kerth and Cunningham, 1997, Meszaros and Doble, 1998) — patterns analysing just one example do not meet this standard.

From our semiotic perspective, patterns of one example do not pose problems, provided it is understood that such patterns analyse a particular design — or how a particular pattern has been applied in a particular case. That is, patterns of one example are critical analyses of actual programs and analyse the signs within a particular text, rather than presenting reusable designs. While a pattern of one example may not provide the reliable design advice of a pattern that has been identified in many different contexts, it can usefully contribute additional information to existing definitions.

For example, patterns in the papers *Documenting Frameworks Using Patterns* (Johnson, 1992) and *Patterns Generate Architectures* (Beck and Johnson, 1994) are all based on a single example — the Hotdraw graphics framework. Other work with this character has begun to appear more regularly, sometimes disguised as new patterns (Hosking et al., 2001, Visser, 2001, Woolf, 2000). The *Pattern Stories Web* also collects “stories” about single uses of existing patterns (Johnston, 2001).

8.5 Further Semiotics of Patterns

The two-level semiotics we have presented (Fig. 4) is sufficient to address questions of the structures, names, and relationships between object-oriented design patterns, and also applies to other kind of patterns — we chose examples from *Design Patterns* simply because it is the best known software patterns collection. We plan to analyse the semiotic structure of object-oriented designs and design patterns in more detail. For example, patterns' designs are partially presented using *class diagrams* (amongst other diagrams and notations); these diagrams are themselves signs, relating a *graphical design* (signifier) to some *class structure* (signified).

We can also consider the pattern itself as participating in further denotative semiosis — patterns are actually written up as book chapters or web pages, so we can consider a *pattern-writeup* as a sign, where the *text of the pattern* from the book is the signifier and a *pattern* is the signified.

Perhaps more powerfully, the discussion of a pattern within a social context will lead to further semiosis. For example, when one programmer suggests using an Observer pattern (the second-level sign) another programmer could reply “Observer? that’s always too slow!”, or “that’s too hard to implement in Java”; or “that could lead to recursive invocations and deadlock”. Each of these replies are alternative signifieds in a third-level, connotative, semiosis for which the second-level sign participates as the signifier (see Fig. 21). As with other types of semiosis, this process can continue indefinitely.

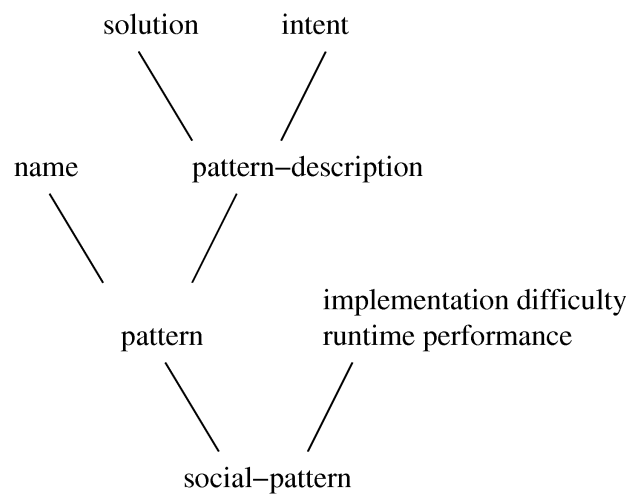


Figure 21: A Connotative Semiotics of Patterns

Finally, patterns are not alone as forms of knowledge about programming. Although we have not yet considered them in depth, data structures and algorithms have a very similar semiotic structure to that we have described for patterns in this article: an algorithm is a named description of a type of signifier (typically code or pseudocode), referring to some computational process, together with the analysis of the algorithm as its signified. Idioms and style rules, architectures, idioms, and cliches may all be amenable to description within this kind of semiotic framework.

9 Related Work

9.1.1 On patterns and the patterns community

Since the publication of *Design Patterns* (Gamma et al., 1994), patterns have become an accepted part of the literature of software engineering. A number of other large-scale patterns texts have been published, some deriving directly from *Design Patterns*, others describing new patterns for the technical design of systems, and still others describing patterns for methodologies or development processes. Yet more individual patterns, or small collections of related patterns have been published in the

Pattern Languages of Program Design book series (Coplien and Schmidt, 1995, Vlissides et al., 1996, Harrison et al., 2000, Martin et al., 1998) or have been presented at various patterns conferences.

Probably the most important documents shaping and recording the development of the patterns “community” are Coplien’s *Software Patterns* (Coplien, 1996) and *Pattern Language for Writer’s Workshops* (Coplien, 2000b); Meszaros and Doble’s *Pattern Language for Pattern Writing* (Meszaros and Doble, 1998) (which neatly sidesteps social restrictions on patterns criticism by employing the pattern form to that end); and the virtual records on the WikiWikiWeb (Cunningham, 2005). Gabriel’s *Patterns of Software* (Gabriel, 1996) and Lea’s *Christopher Alexander: An Introduction* (Lea, 1994) also provide exegeses of Alexander, including an introduction to some of his more recent theorising (Appleton, 1997). In as much as any theory of patterns is presented in these works, it follows Alexander explicitly — the patterns conferences are named “*Pattern Languages of Programming*” (our emphasis) for just this reason.

9.1.2 On analysis of relationships between patterns

Given this flood of primary material there has been surprisingly little analysis of patterns — partly due to an explicit value of the patterns movement to eschew reflection in favour of action (Coplien, 1998). Zimmer provided some early analysis on the relationships between patterns latent within *Design Patterns* (Zimmer, 1994). Many authors of patterns collections proceeded to develop individual schemes of pattern relationships: we have surveyed many of these in previous work (Noble, 1998). These schemes are generally either based upon Alexander-style pattern languages, or are variations of the relationships we analyse in section 6.

Although there have been no complete attempts at restructuring *Design Patterns*, Schmidt et al. (Schmidt et al., 2000, p.509) and Coplien (Coplien, 2000a) have attempted to convert smaller collections of patterns into pattern languages, and Dyson and Anderson have converted the State pattern into a fragment of a pattern language (Dyson and Anderson, 1998).

A more original (and less Alexandrian) analysis of design patterns are the compound (or composite) patterns investigated by Riehle and Vlissides (Riehle, 1997a, Vlissides, 1998). Riehle shows how complex patterns such as Bureaucracy (Riehle, 1998) can be composed from simpler patterns using a role analysis similar to OORAM (Reenskaug, 1996) — essentially the “uses” relationship between patterns. This role analysis also formed the theoretical basis of a catalogue of patterns (Riehle, 1997b). Compared to our work, the role analysis gives more insight into the solutions provided by more complex patterns, but does not address the intents or names of patterns, and is not situated with any conceptual framework.

Tichy produced an early classification of many of the patterns from *Design Patterns* and *Pattern-Oriented Software Architecture* (Tichy, 1997). More recently the *Patterns Almanac* (Rising, 1999) catalogues many patterns published in book form. The classifications underlying these catalogues are generally coarse-grained, and designed to help programmers rather than being based on an underlying theory.

Agerbow and Cornlis (Agerbo and Cornils, 1998) analysed the *Design Patterns* to determine how many patterns were artifacts of programming language, that is, given a sufficiently powerful language how many patterns could be expressed using language features directly, and Gil and Lorenz have developed a similar taxonomy

(Gil and Lorenz, 1998). Coplien and Zhao recently analysed the interactions between patterns and programming languages, in particular where programming language features are not orthogonal (“asymmetrical” in their terminology) (Coplien and Zhao, 2000, Coplien, 2001), and have analysed this using group theory (Zhao and Coplien, 2002).

9.1.3 *On pattern tools and formalisms*

Rather than analyse patterns per se, some work on describing, categorising or recognising patterns has been carried out in order to build tools that support patterns or formalisms that describe them. The earliest work here involved systems that generated code for particular patterns (Budinsky et al., 1996, Soukup, 1994); more recently some support for patterns has been incorporated into experimental CASE tools or programming environments (Eden et al., 1997, Florijn et al., 1997, Mapelsden et al., 2002, Bulka, 2002). Several design notations for patterns have also been proposed — the UML standard now supports patterns by way of parameterised collaborations (Rumbaugh et al., 1998) and a number of more powerful visual techniques have been developed (Lander and Kent, 1998, Sunyé et al., 2000). While several of these systems are useful in practice, in terms of underlying theories of patterns this work has often been completely ad-hoc (e.g. generating whatever code seemed to be required at the time) or has generalised constructs from object-oriented design to represent patterns. The catch is that such approaches miss much of the articulation revealed by our semiotic model: subtleties such as the way the same implementation could support either the Strategy or Decorator pattern, or the multiple implementations of the Adapter pattern, cannot be captured by these approaches.

One alternative to tool support for design patterns is to leverage more direct support in programming languages. Some significant progress has been made along these lines, using Aspect-Oriented Programming to encode pattern implementations separately within program texts (Kiczales et al., 2001). The key advantage of these approaches is that they identify implementations of patterns explicitly within the source code, making it easier for programmers to distinguish patterns from other parts of their programs. Meanwhile, a separate branch of research has focused on applying theory from functional programming to patterns, often focusing on the recursive combination of patterns such as Visitor, (Kühne, 1999, Lorenz, 1997, Visser, 2001). While this work may explain many of the subtleties of implementations of individual patterns, it does not address programmers’ use of patterns to produce and communicate designs, that is, the semiotic aspects of patterns.

Formalisms (generally based upon logic rather than grammars or semiotics) have also been employed to describe patterns. LePUS, for example, describes patterns in the context of a multi-level object model framework (Eden, 2002); other work has used a variety of formal models to capture designs and patterns (e.g. (Mikkonen, 1998)). Again, inasmuch as this work is based on any underlying rationale, they are extensions of concepts drawn from object-oriented design, or naively justified as obviously correct for patterns.

Finally, there are a few tools including Pattern-Lint, that performs static and dynamic analyses of programs to check that they comply with higher level models (Sefika et al., 1996); ArchJava, that can similarly relate a program’s structure to its implementation (Aldrich et al., 2001); and Jacobsen, Nowack, and Kristensen’s conceptual modelling of software artifacts and development processes (Jacobsen et al.,

2000). Although this work is not strictly related to patterns, nor explicitly semiotic, it does take account of the “possibility of lie” (Eco, 1976), that is, it accepts that a design is not the same thing as a program, but rather that one is a (possibly misleading) signifier of the other.

10 Conclusion

In this article, we have described how object-oriented design patterns can be analysed as signs. A pattern-description is a sign where a pattern’s solution is the signifier and the intent is the signified. Then, a pattern is a second-order sign, where the pattern name is the signifier and a pattern-description is the signified.

Treating patterns as signs provides us with an analytic framework that is based on semiotics, rather than logic, mathematics, or mysticism. Using this framework, we have addressed a number of common questions about patterns — explicating patterns that propose similar designs or have similar intents, that have many names or share names, and clarifying the relationships between patterns and the way they function in object-oriented design. Semiotics also allows us to analyse misinterpretations of patterns, and the role of patterns in creating an evolving common vocabulary of program design.

We hope that this framework can provide a platform for future progress in the research and application of design patterns, and for the semiotic analyses of the practices of programming.

Acknowledgements

Thanks to Pippin Barr, Frank Buschmann, Rilla Khaled, Mary Lynn Manns, and Palle Nowack, for their comments on various drafts, to Charles Weir for being around as many of these ideas germinated, and to the anonymous reviewers for their encyclopaedic comments.

References

- Abadi M and Cardelli L (1996) *A Theory of Objects*. Springer-Verlag, New York.
- Agerbo E and Cornils A (1998) *How to preserve the benefits of design patterns*. In *OOPSLA Proceedings*, pages 134-143. ACM.
- Aldrich J, Chambers C and Notkin D (2001) *Component-oriented programming in ArchJava*. In *OOPSLA’01 Workshop on Language Mechanisms for Software Components*. ACM Press, Tampa, Florida.
- Alexander C (1977) *A Pattern Language*. Oxford University Press.
- Alexander C (1979) *The Timeless Way of Building*. Oxford University Press.
- Alexander C (1999) *The origins of pattern theory: The future of the theory, and the generation of a living world*. *IEEE Software*, 16(5), pages 71-82.
- Andersen PB (1992) *Computer semiotics*. *Scandinavian Journal of Information Systems*, 4, pages 3-30.
- Andersen PB (1997) *A Theory of Computer Semiotics*. Cambridge University Press, second edition.

- Andersen PB (2003) *Semiotic models of algorithmic signs*. In Rödiger KH (Ed.), *Algorithmik – Kunst – Semiotik. Hommage für Frieder Nake*, pages 165-211. Synchron, Heidelberg.
- Andersen PB (2005) *Things considered harmful*. In øArgerfalk PJ, Bannon L and Fitzgerald B (Eds.), *Proceedings of ALOIS*2005*.
- Andersen PB, Hasle P and Brandt PA (1997) *Machine semiosis*. In Posner R, Robering K and Sebeok TA (Eds.), *Semiotics: a Handbook about the Sign-Theoretic Foundations of Nature and Culture*, volume 1, pages 548-570. Walter de Gruyter.
- Andersen PB, Holmqvist B and Jensen JF (Eds.) (1993) *The Computer As Medium*. Learning in doing: Social, cognitive and computational perspectives. Cambridge University Press.
- Andersen PB and Nowack P (2002) *Tangible objects: Connecting informational and physical spac*. In Qvortrup L (Ed.), *Virtual Space: The Spatiality of Virtual Inhabited 3D Worlds*, volume 2. Springer-Verlag.
- Appleton B (1997) *On the nature of the nature of order*. Notes on a Presentation given by James O. Coplien to the Chicago Patterns Group. <http://www.enteract.com/~bradapp/docs/NoNoO.html>.
- Barr P (2003) *A Semiotic Model of User-Interface Metaphor*. In *Virtual, Distributed and Flexible Organisations: Studies in Organisational Semiotics - 3*. The 6th International Workshop on Organisational Semiotics: Virtual, Distributed and Flexible Organisations.
- Barthes R (1972) *Mythologies*. Jonathan Cape.
- Bauer L (Ed.) (1998) *Language Myths*. Penguin.
- Bäumer D, Riehle D, Siberski W, Lilienthal C, Megert D, Sylla KH and Züllighoven H (1998) *Values in object systems*. Technical Report 98.10.1, Ubilab, Zurich, Switzerland.
- Bäumer D, Riehle D, Siberski W and Wulf M (1999) *Role object*. In (Harrison et al., 2000).
- Beck K and Cunningham W (1987) *Using pattern languages for object-oriented programs*. Technical report, Tektronix, Inc. Presented at the OOPSLA -87 Workshop on Specification and Design for Object-Oriented Programming.
- Beck K and Johnson R (1994) *Patterns generate architectures*. In *ECOOP Proceedings*.
- Bergman M and Paavola S (2005) *The commens dictionary of Peirce's terms*. <http://www.helsinki.fi/science/commens/dictionary.html>.
- Budinsky FJ, Finnie MA, Vlissides JM and Yu PS (1996) *Automatic code generation from design patterns*. *IBM Systems Journal*, 35(2), pages 151-171.
- Bulka A (2002) *Design pattern automation*. In Noble J and Taylor P (Eds.), *Proceedings of KoalaPlop 2002*, Conferences in Research and Practice in Information Technology. Australian Computer Society.
- Buschmann F, Meunier R, Rohnert H, Sommerlad P and Stal M (1996) *Pattern-Oriented Software Architecture*. John Wiley & Sons.
- Cobley P (Ed.) (2001) *The Routledge Companion to Semiotics and Linguistics*. Routledge, New Fetter Lane, London.
- Cobley P and Jansz L (1997) *Semiotics for Beginners*. Icon Books, Cambridge, England.
- Cockburn A (1998) *Surviving Object-Oriented Projects: A Manager's Guide*. Addison-Wesley.
- Coplien JO (1998) *Pattern value system*. <http://www.c2.com/cgi/wiki?PatternValueSystem>.

- Coplien JO (1994) *A generative development-process pattern language*. In *Pattern Languages of Program Design*. Addison-Wesley.
- Coplien JO (1996) *Software Patterns*. SIGS Management Briefings. SIGS Press.
- Coplien JO (1997) *Idioms and patterns as architectural literature*. *IEEE Software*, 14(1), pages 36-42.
- Coplien JO (2000a) *C++ idioms*. In (Harrison et al., 2000), chapter 10.
- Coplien JO (2000b) *A pattern language for writers' workshops*. In (Harrison et al., 2000).
- Coplien JO (2001) *The future of language: Symmetry or broken symmetry?* In *Proceedings of VS Live 2001*. San Francisco, California.
- Coplien JO and Schmidt DC (Eds.) (1995) *Pattern Languages of Program Design*. Addison-Wesley.
- Coplien JO and Zhao L (2000) *Symmetry and symmetry breaking in software patterns*. In *Proceedings Second International Symposium on Generative and Component Based Software Engineering (GCSE2000)*, pages 373-398.
- Crupi J, Alur D and Malks D (2001) *Core J2EE Patterns*. Prentice Hall PTR.
- Cunningham W () *The wikiwikiweb*. <http://www.c2.com/cgi/wiki>.
- de Saussure F (1916) *Cours de linguistique générale*. V.C. Bally and A. Sechehaye (eds.), Paris/Lausanne.
- Dyson P and Anderson B (1998) *State objects*. In (Martin et al., 1998).
- Easthope A and McGowan K (Eds.) (1992) *A Critical And Cultural Theory Reader*. Allen & Unwin.
- Eco U (1976) *A Theory of Semiotics*. Indiana University Press.
- Eco U (1997) *Kant and the Platypus*. Random House.
- Eden AH (2002) *LePUS: A visual formalism for object-oriented architectures*. In *Sixth World Conference on Integrated Design and Process Technologies*. Society for Design and Process Science.
- Eden AH, Yehudai A and Gil G (1997) *Precise specification and automatic application of design patterns*. In *1997 International Conference on Automated Software Engineering (ASE'97)*.
- Edgar A and Sedgwick P (Eds.) (1999) *Key Concepts in Cultural Theory*. Routledge, New Fetter Lane, London.
- Eliot TS (1962) *Old Possum's Book of Practical Cats*. Faber and Faber, London.
- Ernst MD, Kaplan C and Chambers C (1998) *Predicate dispatching: A unified theory of dispatch*. In *ECOOP Proceedings*, pages 186-211. Springer-Verlag, Brussels.
- Florijn G, Meijers M and van Winsen P (1997) *Tool support for object-oriented patterns*. In *ECOOP Proceedings*, pages 472-468.
- Foote B (1998) *Hybrid vigor and footprints in the snow*. In Martin R, Riehle D and Buschmann F (Eds.), *Pattern Languages of Program Design 3*. Addison-Wesley.
- Foote B and Yoder J (2000) *Big ball of mud*. In Harrison N, Foote B and Rohnert H (Eds.), *Pattern Languages of Program Design*, volume 4, chapter 29, pages 653-692. Addison-Wesley.

- Fowler M (2001) *Value object*. <http://www.martinfowler.org>.
- Fowler M and Scott K (1997) *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley.
- Gabriel RP (1996) *Patterns of Software: Tales from the Software Community*. Oxford University Press.
- Gamma E, Helm R, Johnson RE and Vlissides J (1994) *Design Patterns*. Addison-Wesley.
- Gil JY and Lorenz DH (1998) *Design patterns and language design*. *IEEE Computer*, 31(3), pages 118-120.
- Goguen J (1993) *On notation*. In *TOOLS 10: Technology of Object-Oriented Languages and Systems*, pages 5-10.
- Goguen J (1999) *An introduction to algebraic semiotics, with applications to user interface design*. In Nehaniv C (Ed.), *Computation for Metaphor, Analogy and Agents*, volume 1562 of *LNAI*, pages 242-291. Springer-Verlag.
- Greuter W, Mcneill J, Barrie FR, Burdet HM, Demoulin V, Filgueiras TS, Nicolson DH, Silva PC, Skog JE, Trehane P, Turland NJ and Hawksworth DL (Eds.) (2000) *International Code of Botanical Nomenclature (St. Louis Code). Regnum Vegetabile 131*. Koeltz Scientific Books, Königstein.
- Harrison N, Foote B and Rohnert H (Eds.) (2000) *Pattern Languages of Program Design*, volume 4. Addison-Wesley.
- Henderson-Sellers B (1994) *A BOOK of Object-Oriented Knowledge*. Prentice-Hall.
- Henderson-Sellers B and Edwards JM (1994) *BOOKTWO of Object-Oriented Knowledge: The Working Object*. Prentice-Hall.
- Hillside Inc (2001) *Patterns homepage*. <http://www.hillside.net>.
- Hiraga MK (1994) *Diagrams and metaphors: Iconic aspects in language*. *Journal of Pragmatics*, 22(1), pages 5-21.
- Hosking J, Grundy J and Mugridge W (2001) *Applying and evolving the evolving frameworks pattern language*. In Wallis B and Harrison N (Eds.), *Proceedings of the Second Australasian Conference on Pattern Languages of Programming (KoalaPLoP 2001)*. Victoria University of Wellington.
- Jacobsen EE, Kristensen BB and Nowack P (2000) *Architecture = abstractions over software*. In *TOOLS Pacific*.
- Johnson RE (1992) *Documenting frameworks using patterns*. In *OOPSLA Proceedings*.
- Johnston R (2001) *Patterns stories web*. <http://wiki.cs.uiuc.edu/PatternStories>.
- Kerth NL and Cunningham W (1997) *Using patterns to improve our architectural vision*. *IEEE Software*, 14(1), pages 53-59.
- Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J and Griswold WG (2001) *An overview of AspectJ*. In *ECOOP Proceedings*, pages 327-353.
- Korienek G and Wrensch T (1991) *A Quick Trip To Objectland*. Prentice-Hall.
- Kühne T (1999) *A Functional Pattern System for Object-Oriented Design*, volume 47 of *Forschungsergebnisse zur Informatik*. Verlag Dr. Kovac.
- Lander A and Kent S (1998) *Precise visual specification of design patterns*. In *ECOOP Proceedings*, pages 114-134.

- Lea D (1994) *Christopher Alexander: An introduction for object-oriented designers*. ACM Software Engineering Notes.
- Lehrmann Madsen O, Møller-Pedersen B and Nygaard K (1993) *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley.
- Liu K (2000) *Semiotics in Information Systems Engineering*. Cambridge University Press.
- Liu K, Clarke RJ, Andersen PB and Stamper RK (Eds.) (2002) *Organizational Semiotics: Evolving a Science of Information Systems.*, volume 227 of *IFIP Conference Proceedings*. Kluwer.
- Lorenz DH (1997) *Tiling design patterns — a case study*. In *OOPSLA Proceedings*.
- Mapelsden D, Hosking J and Grundy J (2002) *Design pattern modelling and instantiation using DPML*. In Noble J and Potter J (Eds.), *In Proc. Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Conferences in Research and Practice in Information Technology. Australian Computer Society.
- Martin RC, Riehle D and Buschmann F (Eds.) (1998) *Pattern Languages of Program Design*, volume 3. Addison-Wesley.
- Meszaros G and Doble J (1998) *A pattern language for pattern writing*. In (Martin et al., 1998).
- Meyer B (1988) *Object-oriented Software Construction*. Prentice Hall.
- Meyer B (1997) *Object-Oriented Software Construction*. Prentice Hall PTR, second edition.
- Mikkonen T (1998) *Formalizing design patterns*. In *International Conference on Software Engineering (ICSE)*, pages 115-124.
- Noble J (1998) *Classifying relationships between object-oriented design patterns*. In *Australian Software Engineering Conference (ASWEC)*, pages 98-107.
- Noble J (2000) *Basic relationship patterns*. In Harrison N, Foote B and Rohnert H (Eds.), *Pattern Languages of Program Design 4*, chapter 6, pages 73-94. Addison-Wesley.
- Noble J and Biddle R (2002) *Patterns as signs*. In *ECOOP Proceedings*.
- Noble J, Biddle R and Tempero E (2002) *Metaphor and metonymy in object-oriented design patterns*. In *Proceedings of Australian Computer Science Conference (ACSC)*. Australian Computer Society.
- Peirce CS (1934-1948) *Collected Papers*. four volumes. Harvard University Press.
- Petre M, Blackwell A and Green T (1997) *Cognitive questions in software visualisation*. In Stasko J, Domingue JB, Price BA and Brown M (Eds.), *Software Visualization: Programming as a Multimedia Experience*. M.I.T. Press.
- Raymond E and Steele GL (1993) *The New Hacker's Dictionary*. MIT Press, second edition.
- Reenskaug T (1996) *Working with Objects: The OOram Software Engineering Method*. Manning Publications.
- Riehle D (1997a) *Composite design patterns*. In *ECOOP Proceedings*.
- Riehle D (1997b) *A role based design pattern catalog of atomic and composite patterns structured by pattern purpose*. Technical Report 97-1-1, Ubilab, Zurich, Switzerland.
- Riehle D (1998) *Bureaucracy*. In (Martin et al., 1998).

- Riehle D and Züllighoven H (1996) *Understanding and using patterns in software development. Theory and Practice of Object Systems*, 2(1), pages 3-13.
- Rising L (1999) *The Pattern Almanac 2000*. Addison-Wesley.
- Rowling JK (1997) *Harry Potter and the Philosopher's Stone*. Bloomsbury.
- Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorensen W (1991) *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey.
- Rumbaugh J, Jacobson I and Booch G (1998) *The Unified Modeling Language Reference Manual*. Addison-Wesley.
- Schmidt D, Stal M, Rohnert H and Buschmann F (2000) *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons.
- Sebeok TA (2001) *Nonverbal communication*. In (Cobley, 2001), chapter 1.
- Sefika M, Sane A and Campbell RH (1996) *Monitoring compliance of a software system with its high-level design models*. In *Proceedings of the 18th Int'l Conf. on Software Eng., (ICSE-18)*.
- Soukup J (1994) *Taming C++: Pattern Classes and Persistence for Large Projects*. Addison-Wesley.
- Stoll NR (Ed.) (1964) *International code of zoological nomenclature*. International Commission on Zoological Nomenclature, 2nd edition.
- Sunyé G, Guennec AL and Jézéquel JM (2000) *Design pattern application in UML*. In *ECOOP Proceedings*.
- Taivalsaari A (1993) *Object-oriented programming with modes*. *Journal of Object-Oriented Programming*, 6(3), pages 25-32.
- Tichy WF (1997) *A catalogue of general-purpose software design patterns*. In *TOOLS USA 1997*.
- Ungar D and Smith RB (1991) *SELF: the Power of Simplicity*. *Lisp And Symbolic Computation*, 4(3).
- Visser J (2001) *Visitor combination and traversal control*. In *OOPSLA Proceedings*, pages 270-282.
- Vlissides J (Ed.) (1998) *Pattern Hatching: Design Patterns Applied*. Addison-Wesley.
- Vlissides JM, Coplien JO and Kerth NL (Eds.) (1996) *Pattern Languages of Program Design*, volume 2. Addison-Wesley.
- Wirfs-Brock R, Wilkerson B and Wiener L (1990) *Designing Object-Oriented Software*. Prentice-Hall.
- Woolf B (1998) *Null object*. In (Martin et al., 1998).
- Woolf B (2000) *Object recursion*. In (Harrison et al., 2000).
- Zhao L and Coplien JO (2002) *Symmetry in class and type hierarchy*. In Noble J and Potter J (Eds.), *In Proc. Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Conferences in Research and Practice in Information Technology. Australian Computer Society.
- Zimmer W (1994) *Relationships between design patterns*. In *Pattern Languages of Program Design*. Addison-Wesley.

About the Authors

James Noble is Professor of Computer Science and Software Engineering at Victoria University of Wellington, New Zealand. His research centres around software design. This includes the design of the users' interface, the parts of software that users have to deal with every day, and the programmers' interface, the internal structures and organisations of software that programmers see only when they are designing, building, or modifying software. His research in both of these areas is coloured by his longstanding interest in object-oriented approaches to design.

Robert Biddle is Professor of Human-Computer Interaction at Carleton University in Ottawa, Canada. His research range concerns the design of software from the human point of view, and includes the concerns of both programmers and users. Robert is particularly interested in ways Computing can learn from the Humanities, and his current work involves diverse multi-disciplinary programmes. Robert was a British Commonwealth Scholar, and for many years worked in New Zealand, where he held appointments in both Computer Science and Information Systems.

Ewan Tempero is an Associate Professor in the Department of Computer Science at The University of Auckland, New Zealand. He primarily teaches in the Software Engineering specialisation of the Bachelor of Engineering Degree. His research currently focuses on understanding the relationship between code structure and quality attributes such as maintainability, testability, and reusability.