VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

Patterns as Signs

James Noble and Robert Biddle

Technical Report CS-TR-01/16
November 2001

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Patterns as Signs

James Noble and Robert Biddle

Technical Report CS-TR-01/16
November 2001

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. A number of open research problems remain regarding patterns, however, including the differences between patterns, variant forms of common patterns, the naming of patterns, the organisation of collections of patterns, and the relationships between patterns. 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. Considering patterns as signs can address many of these common questions regarding design patterns, to assist both programmers using patterns and authors writing them.

Publishing Information

A later version of this paper was to be presented at the European Conference on Object-Oriented Programming, Malaga, Spain, 2002.

Author Information

The authors members of the academic staff in the School of Mathematical and Computing Sciences and the School of Information Management at Victoria University of Wellington.

Patterns as Signs

James Noble and Robert Biddle

Computer Science
Victoria University of Wellington
New Zealand
kjx@mcs.vuw.ac.nz

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. A number of open research problems remain regarding patterns, however, including the differences between patterns, variant forms of common patterns, the naming of patterns, the organisation of collections of patterns, and the relationships between patterns. 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. Considering patterns as signs can address many of these common questions regarding design patterns, to assist both programmers using patterns and authors writing them.

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*" [44, p.3]. Designers can incorporate patterns into their program to address general problems in the structure of their programs' designs, in a similar way that algorithms or data structures are incorporated into programs to solve particular computational or storage problems. A growing body of literature catalogues patterns for object-oriented design, including reference texts such as *Design Patterns* [44] or *Pattern-Oriented Software Architecture* [17, 70], and patterns compendia such as the *Pattern Languages of Program Design* series [28, 78, 57, 48].

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 Adapter); how can several different patterns have the same name (such as Prototype); and how can the relationships between patterns best be characterised.

In this paper, 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 [35]. When semiotics began in the early years of the last century, most work was concerned with conventional signs — first speech, and then writing. 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) [71]. One of the avowed values of the design patterns movement is to treat "patterns as literature" [51, 24]; our semiotic approach builds on this idea by applying techniques from the study of literature and culture to programs and patterns.

This paper 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, and then section 5 addresses a number of open questions in the analysis of design patterns, showing how the semiotic approach can cast some light upon these problems. Section 6 discusses the ramifications of our approach more broadly, section 7 places this approach in the context of other work organising and theorising patterns, and other work on the semiotics of information processing, and finally, section 8 concludes the paper and 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 [44, 17]. Patterns were developed by an architect, Christopher Alexander [54], to describe techniques for town planning, architectural designs, and building construction techniques, and described in Alexander's *A Pattern Language • Towns, Buildings, Construction* [4, 3, 5]. Design patterns were first applied to software by Kent Beck and Ward Cunningham to describe user interface design techniques [14, 51], 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 [63, 77].

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 [23, 58, 66]. 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, and the important considerations and consequences of using the pattern, a sample implementation of the pattern's solution, and references to known uses of the pattern and to other patterns to which it is related.

Moreso than other forms of writing about software, patterns are self-consciously "*literature*" about software. The patterns "PLoP" conference series, for example, has modelled itself on some parts of the creative writing community. At PLoP conferences, for example, papers are workshopped to improve their *expression* (as against than their *content*), rather than being presented to a passive audience [26]. The patterns movement catchphrase *"the aggressive disregard for innovation"*[1] again encapsulates this idea: the focus is on the literary expression of existing tested ideas, rather than the advocacy of new idiosyncrasies.

[1] Attributed to Thomas J. "Tad" Peckish by Brian Foote [41].

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 [32] is the study of signs in society; where a *sign* is *"something standing for something else"* [35]. Saussure was a linguist, so we will mostly use examples from language in this section, although semiotics has now been applied to a wide range of different kind of signs.

The key idea underlying semiotics is the *sign*, shown in Fig. 1.

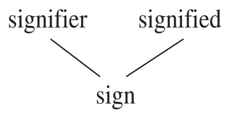

Fig. 1. Saussure's Sign

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[2].

One important principle from Saussure is the "*arbitrariness of the sign"*, that is, that the relationship between signifier and signified can be an arbitrary one. There is no compelling reason why the colour red should be associated with the signifier (name) "red" rather than the signifier "yellow" or signifier "blue". Yellow, for example, could be just as well be expressed by "jaune" or "gelb" or "glonko", provided all participants in the communication knew that this was the signifier for yellow.

The arbitrariness of signs is compounded because signifiers and signifieds are not defined absolutely: rather they are only distinguishable relatively by *difference* from each other. Saussure defines a value (such as a five franc piece) as something which can be *exchanged* for something *different* (a loaf of bread) or *compared* with something *similar* (a ten franc piece). In this way, a signifier may be compared with another signifier, or a signified with a signified, or a signifier may be exchanged with a signified when taken as a sign.

[2] 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 this paper as it suffices for our analysis [35].

In spoken English, for example, there are no absolute definitions of the way the signifiers "rid", "red", and "reed" should be pronounced: the pronunciations blend into one another and what is "red" pronounced with one accent may be "rid" with another. Other spoken languages function in this way. Furthermore, any utterance (*token* is the semiotic term) of a given word will differ slightly from any other utterance, even from the same speaker, although all will be understood as the same word (or *type*). This is also true for signifieds: the colours pink, red, and brown, for example, differ according to their saturation, but we can't say for sure where pink ends and red begins — or rather, such definitions are relative and arbitrary. Another affect of the arbitrariness of signs is that signifiers and signifieds are not uniquely related: signs are individual, rather than their component parts. So in spoken English for example, the same signifier is part of the sign for the colour red and the past tense of verb "to read", and some particular instance of the colour red could also be spoken of by words such as "maroon", "crimson", or even "orange" or "brown".

Signs carry meaning in communication because the participants understand the structures of the signs that make up the messages exchanged between them — that is, the relations of difference between signifiers and between signifieds. Saussure introduces the term *langue* to signify the entire underlying abstract structure of a system of signs — a repertoire of possibilities (or differences) from which a language community can construct messages. Every participant in a communication tacitly shares the same *langue*. In contrast to the overarching langue, the speech acts or sign instances making up a particular communication (the subset of the langue actually used in any given message) is termed the *parole*. A *text* — a given instance of *parole* — a single utterance, a sentence, a conference paper — will be made up of a series of signs taken from the *langue*, according to the rules by which it operates: the meaning of the whole message is produced by the interdependencies between the meanings of the individual signs.

This section has provided only a brief introduction to semiotics, which is capable of much more complex and subtle analyses than those we have presented here [35, 7, 18]. We have kept this presentation to the minimum necessary to support our account of design patterns. For similar reasons, most of our examples of object-oriented patterns are taken from Gamma et. al.'s *Design Patterns* [44] because this is the best known collection of patterns, although our approach is applicable to other kinds of patterns.

4 Patterns as Signs

In the classic definition, a pattern is a *"solution to a problem in a context"* [54, 49]. A 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* [3]) 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 which uses the pattern, the elements corresponding to the pattern's solution can literally be pointed to in a listing of the programs 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.

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 [68] 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 this diagram as an application of the Composite pattern, and thus bring their knowledge of that pattern

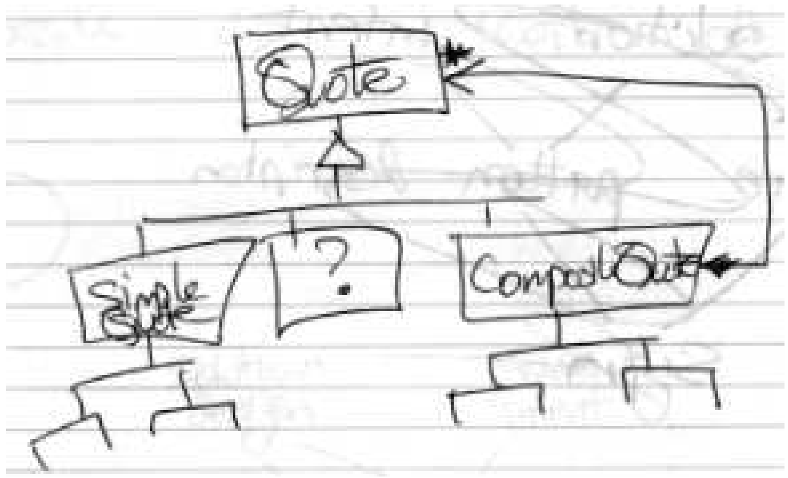

Fig. 2. A sketch of a class diagram for part of a sales quotation system.

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 [44].

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 cupboards and Composite patterns) to make sense of those concrete features.

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

> ***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.

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

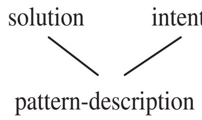

Fig. 3. A pattern description as a sign.

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 which 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 language with a common vocabulary that programmers can use to talk about design [54, 44, 23, 58]. 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, in such a way that one sign is a component of the other. It is a *denotative* system because the second-order sign names the first-order sign[3] [18].

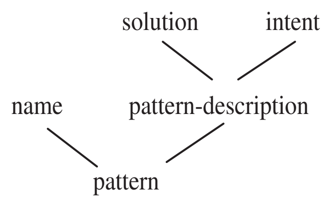

Fig. 4. A pattern as a second-order sign

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 signified. 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

The 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. The contribution of this paper is in the semiotic explanation of the answers to these questions.

[3] This is in contrast to many other semiotic systems, where a (denotative) first-order sign is the signifier of a (connotative) second-order sign. 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 6.3.

5.1 Questions of Pattern Descriptions

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

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 from *Design Patterns*. How, then, can they be different patterns? Would we not be better of with a single pattern encompassing both State and Strategy? [1].

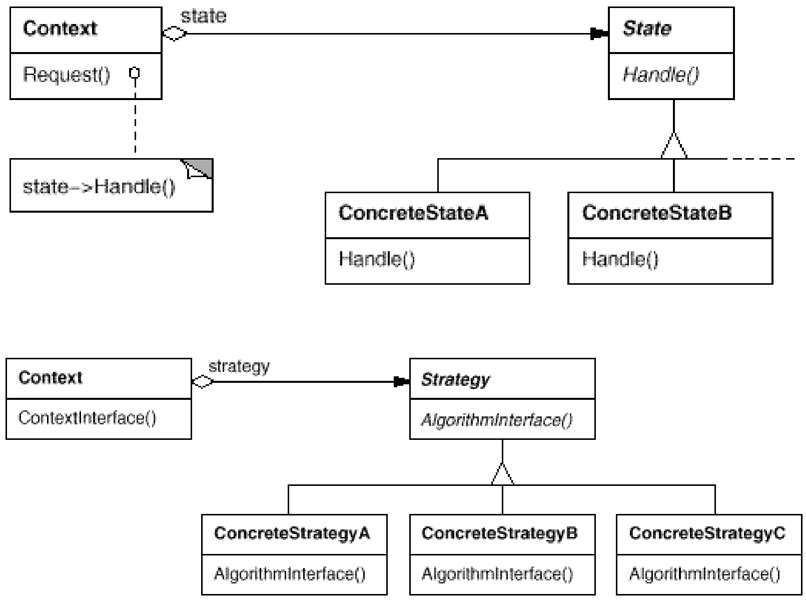

Fig. 5. *Design Patterns* State and Strategy pattern structure diagrams [44].

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.

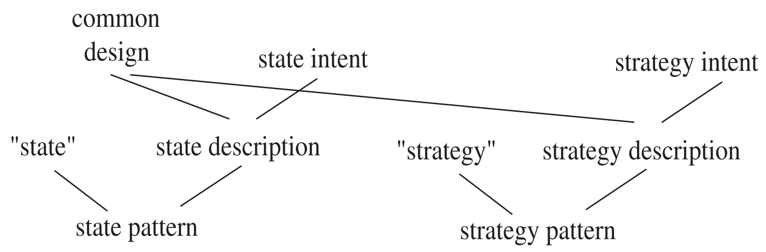

Fig. 6. State and Strategy Patterns

How can one pattern have more than one implementation? Sometimes the text of a pattern describes more than one implementation. For example, the Adaptor pattern describes four separate kinds of Adaptors — 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.

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 design (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* [18]; 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 (*"Do you mean a Class Adaptor or an Object Adaptor?"*) [36]. 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

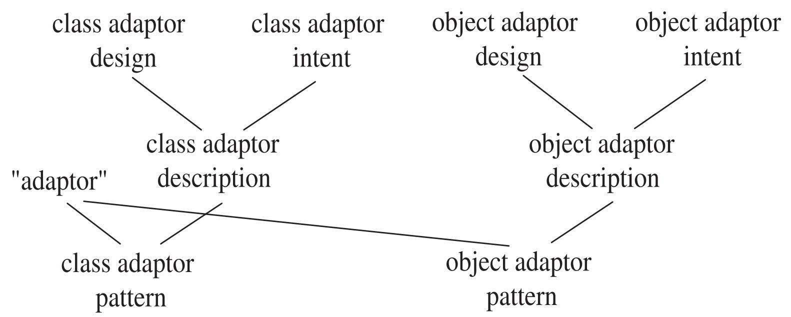

Fig. 7. Polysemy in the Adaptor pattern

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).

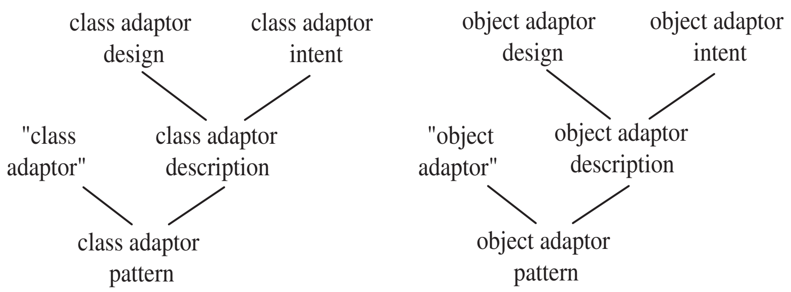

Fig. 8. Disambiguated Adaptor patterns

How can one pattern solve two or more problems? Complementing those patterns which have multiple solutions, some patterns are described as solving multiple problems with a single design. The best example here is the Proxy pattern: the *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 which 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).

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

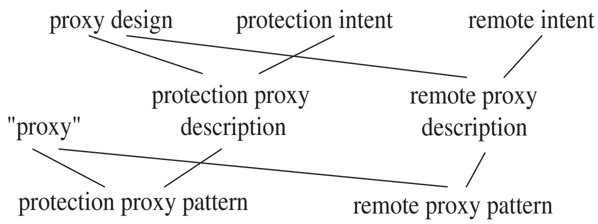

Fig. 9. Multipurpose patterns

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.

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.

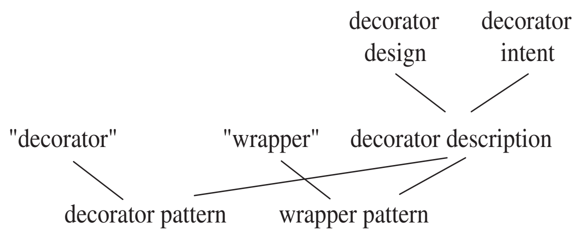

Fig. 10. Patterns with synonyms

One case where this has not happened has been with the name “Wrapper”. *Design Patterns* gives both the Adapter and Decorator patterns the synonym “Wrapper”; however the name “Wrapper” is still in general use both for Adapters, 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.

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 patterns[4]. For example, in the *Patterns Almanac* [67] 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* [22]; a similar pattern from Cockburn’s *Surviving Object-Oriented Projects* [20]: 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 [30] includes a pattern named “Value Object” which is quite different from existing patterns called “Value Object” [42, 52].

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* [67] 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 and Cockburn’s Prototype pattern. Furthermore, a single “pattern” can be described in a number of 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 struc-

[4] This problem was identified by Linda Rising.

ture 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.

5.3 Relationships Between Patterns

Semiotics, being fundamentally concerned with the difference between signifiers, signifieds, and signs, can also help define the relationships between patterns [81, 58, 60]. 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.

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 you must apply one pattern as part of applying the other pattern — for this reason, the larger pattern is often called a *compound* pattern [63, 77].

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).

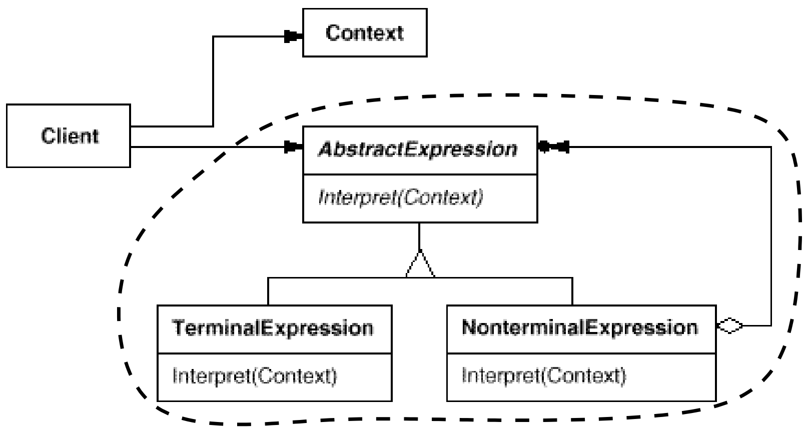

Fig. 11. *Design Patterns* Interpreter structure showing Composite substructure [44]

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.

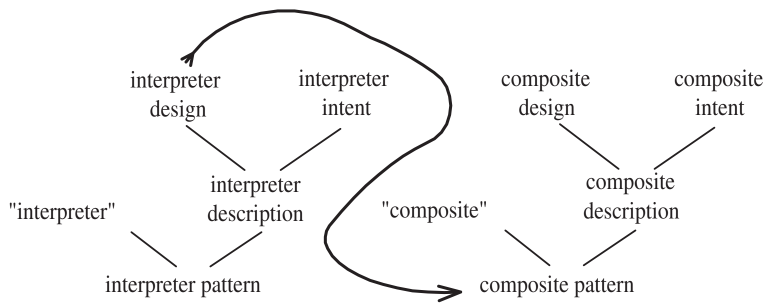

Fig. 12. Composite and Interpreter Patterns

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, possibility requiring the object to be changed to be aware of the extension [44, 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).

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).

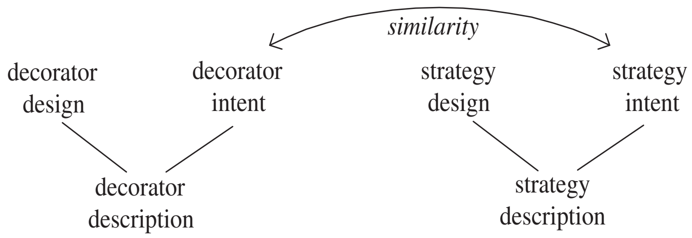

Fig. 13. Decorator and Strategy Patterns

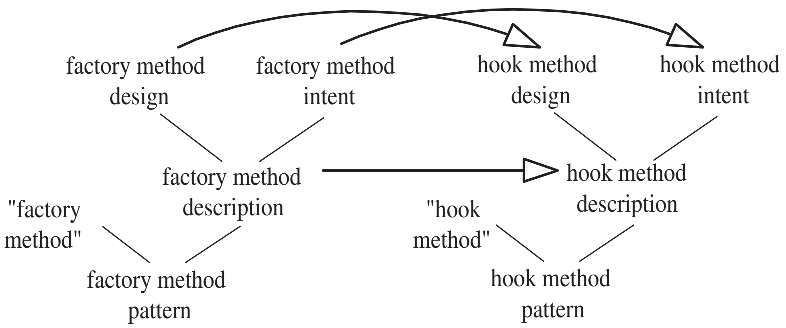

Fig. 14. Factory and Hook Methods

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.

6 Discussion

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

6.1 Misinterpreting Patterns

One of the biggest challenges in documenting patterns is to avoid their misinterpretation[5] — 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, and private mental concepts — 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.

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 [35] 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 accidently 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" [36] — "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 [61]. In the patterns community, the shepherding and workshopping of patterns at PLoP conferences provides an explicit forum for these negotiations, and thus helps to manage misinterpretation of patterns [26].

6.2 Patterns and Pattern Languages

Alexander's architectural patterns are contained within a larger structure of patterns known as a *pattern language* [4, 3] — 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 which address subproblems raised by that solution. The language begins with an *initial pattern* (like a grammar's start symbol) addressing a large scale

[5] This observation is due to Frank Buschmann.

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 windowsills [3].

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?"*[6].

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 [70]. 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 [19, 35, 36]. Indeed, most pattern books are structured this way, to a greater or lesser extent (*Design Patterns* even describes itself as an "encyclopedia" (sic) [44, 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 [60].

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 5.3) a structure like a pattern language is created in a localised part of the pattern 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 or removed from a encyclopædia without affecting the integrity of the encyclopædia. Several later patterns (such as Null Object [79], Value Object [52, 12, 42], and Role Object [13]) have effectively been added to the

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

vocabulary originated by *Design Patterns* while some (such as Builder or Interpreter) have almost fallen out of common use.

6.3 A More Detailed 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 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. The discussion of a pattern within a social context illustrates another way a pattern can be treated as a sign: when a pattern is discussed it raises other connotations in participants in the conversation ("Observer? that's always too slow!").

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 paper: an algorithm is a named description of a concrete signifier (typically code or pseudocode) 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.

7 Related Work

On patterns and the patterns community Since the publication of *Design Patterns* [44], 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 [28, 78, 48, 57] 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* [23] and *Pattern Language for Writer's Workshops* [26]; Meszaros and Doble's *Pattern Language for Pattern Writing* [58] (which neatly sidesteps social restrictions on patterns criticism by employing the pattern form to that end); and the virtual records on the WikiWikiWeb [31]. Gabriel's *Patterns of Software* [43] and Lea's *Christopher Alexander: An Introduction* [54] also provide exegeses of Alexander, including an introduction to some of his more recent theorising [11]. 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.

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 reflexion in favour of action [21]. Zimmer provided some early analysis on the relationships between patterns latent within *Design Patterns* [81]. Many authors of patterns collections proceeded to develop individual schemes of pattern relationships: we have surveyed many of these in previous work [60]. These schemes are generally either based upon Alexander-style pattern languages, or are variations of the relationships we analyse in section 5.3.

Although there have been no complete attempts at restructuring *Design Patterns*, Schmidt et. al. [70, p.509] and Coplien [25] 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 [33].

A more original (and less Alexandrian) analysis of design patterns are the compound (or composite) patterns investigated by Riehle and Vlissides [63, 77]. Riehle shows how complex patterns such as Bureaucracy [65] can be composed from simpler patterns using a role analysis similar to OORAM [62] — essentially the "uses" relationship between patterns. This role analysis also formed the theoretical basis of a catalogue of patterns [64]. 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* [75]. More recently the *Patterns Almanac* [67] 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 [1] 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 [45]. 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) [29, 27], and have analysed this using group theory [80]. 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, [52, 55, 76]. 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.

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 [15, 73]; more recently some support for patterns has been incorporated into experimental CASE tools or programming environments [37, 40, 56, 16]. Several design notations for patterns have also been proposed — the UML standard now supports patterns by way of parameterised collaborations [69] and a number of more powerful visual techniques have been developed [53, 74]. 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 Adaptor pattern, cannot be captured by these approaches.

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 [38]; other work has used a variety of formal models to capture designs and patterns (e.g. [59]). 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.

Pattern-Lint performs static and dynamic analyses of programs to check that they comply with higher level models [72], ArchJava can similarly relate a program's structure to its implementation [2], and Jacobsen, Nowack, and Kristensen have applied conceptual modelling to software artifacts and development processes [50]. Although this work is not strictly related to patterns, nor explicitly semiotic, it does take account of the "possibility of lie" [35], that is, it accepts that a design not the same thing as a program, but a (possibly incorrect) signifier.

On semiotics Although semiotics has adapted 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 [7, 9, 8, 10]. Andersen also argued for a semiotic approach to information systems, rather than relying solely upon generative grammars or logic [6]. Gougen has established Algebraic Semiotics, also primarily concerned with user interface design [47] and design notations [46], focusing on formal systems.

Regarding semiotics more generally, *Semiotics for Beginners* is quite approachable (with many pictures!) [19] and a variety of readers and companions are often intelligible even to readers with technical backgrounds [34, 39, 18]. The semiotics used in this paper is a very small part of that proposed by Eco [35].

8 Conclusion

In this paper, 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 a name is a 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, mysticism, or a metaphor without a name. 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. 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.

9 Acknowlegements

Thanks to Frank Buschmann, Mary Lynn Manns, Palle Nowack, and Ewan Tempero 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 encyclopædic comments.

References

1. Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In *OOPSLA Proceedings*, pages 134–143. ACM, 1998.
2. Jonathan Aldrich, Craig Chambers, and David Notkin. Component-oriented programming in ArchJava. In *OOPSLA'01 Workshop on on Language Mechanisms for Software Components*. ACM Press, Tampa, Florida, October 2001.
3. Christopher Alexander. *A Pattern Language*. Oxford University Press, 1977.
4. Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.
5. Christopher Alexander. The origins of pattern theory: The future of the theory, and the generation of a living world. *IEEE Software*, 16(5):71–82, September 1999.
6. Peter Bøgh Andersen. Computer semiotics. *Scandinavian Journal of Information Systems*, 4:3–30, 1992.
7. Peter Bøgh Andersen. *A Theory of Computer Semiotics*. Cambridge University Press, second edition, 1997.
8. Peter Bøgh Andersen, Per Hasle, and Per Aage Brandt. Machine semiosis. In Roland Posner, Klaus Robering, and Thomas A. Sebeok, editors, *Semiotics: a Handbook about the Sign-Theoretic Foundations of Nature and Culture*, volume 1, pages 548–570. Walter de Gruyter, 1997.
9. Peter Bøgh Andersen, Berit Holmqvist, and Jens F. Jensen, editors. *The Computer As Medium*. Learning in doing: Social, cognitive and computational perspectives. Cambridge University Press, 1993.
10. Peter Bøgh Andersen and Palle Nowack. Tangible objects: Connecting informational and physical spac. In L. Qvortrup, editor, *Virtual Space: The Spatiality of Virtual Inhabited 3D Worlds*, volume 2. Springer-Verlag, 2002.

11. Brad Appleton. 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`, August 1997.
12. Dirk Bäumer, Dirk Riehle, Wolf Siberski, Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, , and Heinz Züllighoven. Values in object systems. Technical Report Technical Report 98.10.1, Ubilab, Zurich, Switzerland, 1998.
13. Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. Role object. In Harrison et al. [48].
14. Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. Technical report, Tektronix, Inc., 1987. Presented at the OOPSLA-87 Workshop on Specification and Design for Object-Oriented Programming.
15. F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.
16. Andy Bulka. Design pattern automation. In James Noble and Paul Taylor, editors, *Proceedings of KoalaPlop 2002*, To Appear in Conferences in Research and Practice in Information Technology. Australian Computer Society, 2002.
17. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.
18. Paul Cobley, editor. *The Routledge Companion to Semiotics and Linguistics*. Routledge, New Fetter Lane, London, 2001.
19. Paul Cobley and Litza Jansz. *Semiotics for Beginners*. Icon Books, Cambridge, England, 1997.
20. Alistair Cockburn. *Surviving Object-Oriented Projects: A Manager's Guide*. Addison-Wesley, 1998.
21. James O. Coplien. Pattern value system. `http://www.c2.com/cgi/wiki?PatternValueSystem`.
22. James O. Coplien. A generative development-process pattern language. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.
23. James O. Coplien. *Software Patterns*. SIGS Management Briefings. SIGS Press, 1996.
24. James O. Coplien. Idioms and patterns as architectural literature. *IEEE Software*, 14(1):36–42, January 1997.
25. James O. Coplien. C++ idioms. In Harrison et al. [48], chapter 10.
26. James O. Coplien. A pattern language for writer's workshops. In Harrison et al. [48].
27. James O. Coplien. The future of language: Symmetry or broken symmetry? In *Proceedings of VS Live 2001*, San Francison, California, January 2001.
28. James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.
29. James O. Coplien and Liping Zhao. Symmetry and symmetry breaking in software patterns. In *Proceedings Second International Symposium on Generative and Component Based Software Engineering (GCSE2000)*, pages 373–398, 2000.
30. John Crupi, Deepak Alur, and Dan Malks. *Core J2EE Patterns*. Prentice Hall PTR, 2001.
31. Ward Cunningham. The wikiwikiweb. `http://www.c2.com/cgi/wiki`.
32. Ferdinand de Saussure. *Cours de linguistique générale*. V.C. Bally and A. Sechehaye (eds.), Paris/Lausanne, 1916.
33. Paul Dyson and Bruce Anderson. State objects. In Martin et al. [57].
34. Anthony Easthope and Kate McGowan, editors. *A Critical And Cultural Theory Reader*. Allen & Unwin, 1992.
35. Umberto Eco. *A Theory of Semiotics*. Indiana University Press, 1976.
36. Umberto Eco. *Kant and the Platypus*. Random House, 1997.

37. A. H. Eden, A. Yehudai, and G. Gil. Precise specification and automatic application of design patterns. In *1997 International Conference on Automated Software Engineering (ASE'97)*, 1997.
38. Amnon H. Eden. LePUS: A visual formalism for object-oriented architectures. In *Sixth World Conference on Integrated Design and Process Technologies*. Society for Design and Process Science, June 2002.
39. Andrew Edgar and Peter Sedgwick, editors. *Key Concepts in Curtural Theory*. Routledge, New Fetter Lane, London, 1999.
40. Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In *ECOOP Proceedings*, pages 472–468, 1997.
41. Brian Foote. Hybrid vigor and footprints in the snow. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
42. Martin Fowler. Value object. `http://www.martinfowler.org`, 2001.
43. Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.
44. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
45. Joseph (Yossi) Gil and David H. Lorenz. Design patterns and language design. *IEEE Computer*, 31(3):118–120, March 1998.
46. Joseph Gougen. On notation. In *TOOLS 10: Technology of Object-Oriented Languages and Systems*, pages 5–10, 1993.
47. Joseph Gougen. An introduction to algebraic semiotics, with applications to user interface design. In Chrystopher Nehaniv, editor, *Computation for Metaphor, Analogy and Agents*, volume 1562 of *LNAI*, pages 242–291. Springer-Verlag, 1999.
48. Neil Harrison, Brian Foote, and Hans Rohnert, editors. *Pattern Languages of Program Design*, volume 4. Addison-Wesley, 2000.
49. Hillside Inc. Patterns homepage. `http://www.hillside.net`, 2001.
50. Eydun Eli Jacobsen, Bent Bruun Kristensen, and Palle Nowack. Architecture = abstractions over software. In *TOOLS Pacific*, 2000.
51. Norman L. Kerth and Ward Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, 14(1):53–59, January 1997.
52. Thomas Kühne. *A Functional Pattern System for Object-Oriented Design*, volume 47 of *Forschungsergbnisse zur Informatik*. Verlag Dr. Kovač, 1999.
53. Anthony Lander and Stuart Kent. Precise visual specification of design patterns. In *ECOOP Proceedings*, pages 114–134, 1998.
54. Doug Lea. Christopher alexander: An introduction for object-oriented designers. *ACM Software Engineering Notes*, January 1994.
55. David H. Lorenz. Tiling design patterns — a case study. In *OOPSLA Proceedings*, 1997.
56. D. Mapelsden, J. Hosking, and J. Grundy. Design pattern modelling and instantiation using DPML. In James Noble and John Potter, editors, *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, 2002.
57. Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1998.
58. Gerard Meszaros and Jim Doble. A pattern language for pattern writing. In Martin et al. [57].
59. Tommi Mikkonen. Formalizing design patterns. In *International Conference on Software Engineering (ICSE)*, pages 115–124, 1998.
60. James Noble. Classifying relationships between object-oriented design patterns. In *Australian Software Engineering Conference (ASWEC)*, pages 98–107, 1998.

61. M. Petre., A. F. Blackwell, and T.R.G. Green. Cognitive questions in software visualisation. In John Stasko, John B. Domingue, Blaine A. Price, and Marc Brown, editors, *Software Visualization: Programming as a Multimedia Experience*. M.I.T. Press, 1997.
62. Trygve Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.
63. Dirk Riehle. Composite design patterns. In *ECOOP Proceedings*, 1997.
64. Dirk Riehle. A role based design pattern catalog of atomic and composite patterns structured by pattern purpose. Technical Report 97-1-1, UbiLabs, 1997.
65. Dirk Riehle. Bureaucracy. In Martin et al. [57].
66. Dirk Riehle and Heinz Züllighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
67. Linda Rising. *The Pattern Almanac 2000*. Addison-Wesley, 1999.
68. J. K. Rowling. *Harry Potter and the Philospher's Stone*. Bloomsbury, 1997.
69. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
70. Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2000.
71. Thomas A. Sebeok. Nonverbal communication. In Cobley [18], chapter 1.
72. Mohlalefi Sefika, A. Sane, and R. H. Campbell. 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)*, 1996.
73. Jiri Soukup. *Taming C++: Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994.
74. Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design pattern application in UML. In *ECOOP Proceedings*, 2000.
75. Walter F. Tichy. A catalogue of general-purpose software design patterns. In *TOOLS USA 1997*, 1997.
76. Joost Visser. Visitor combination and traversal control. In *OOPSLA Proceedings*, pages 270–282, 2001.
77. John Vlissides, editor. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.
78. John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.
79. Bobby Woolf. Null object. In Martin et al. [57].
80. Liping Zhao and James O. Coplien. Symmetry in class and type hierarchy. In James Noble and John Potter, editors, *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, 2002.
81. Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*. Addison-Wesley, 1994.