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

School of Mathematical and Computing Sciences
Computer Science

Metaphor and Metonymy in Object-Oriented Design Patterns

James Noble and Robert Biddle and Ewan Tempero

Technical Report CS-TR-01/7
August 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

Metaphor and Metonymy in Object-Oriented Design Patterns

James Noble and Robert Biddle and Ewan Tempero

Technical Report CS-TR-01/7
August 2001

Abstract

The key principle of object-oriented design is that each program object should correspond to an object in the real world, that is to say, a program is a metaphor for the world. More advanced object-oriented designs, such as many of Gamma et. al.'s Design Patterns, are not directly metaphorical: State objects, Strategy objects and Visitor objects, for example, do not correspond to objects in the real world. We show how these patterns, and other similar designs, can be understood as metonymy, rather than metaphor, that is, they are based on an attribute, cause, or effect, rather than being based on something in the world, in terms of Jakobson and Lodge's typology. Understanding how both metaphor and metonymy can be used in design can illustrate how design patterns work alongside more traditional object-oriented modelling to produce designs that are accurate, flexible, and better understood.

Publishing Information

Author Information

James Noble, Robert Biddle, and Ewan Tempero lecture in Computer Science at Victoria Universtiy of Wellington.

Metaphor and Metonymy in Object-Oriented Design Patterns

James Noble **Robert Biddle** **Ewan Tempero**

Computer Science
Victoria University of Wellington
New Zealand
{kjx,robert,ewan}@mcs.vuw.ac.nz

Abstract

The key principle of object-oriented design is that each program object should correspond to an object in the real world, that is to say, a program is a *metaphor* for the world. More advanced object-oriented designs, such as many of Gamma et. al.'s *Design Patterns*, are not directly metaphorical: State objects, Strategy objects and Visitor objects, for example, do **not** correspond to objects in the real world. We show how these patterns, and other similar designs, can be understood as *metonymy*, rather than metaphor, that is, they are based on an *attribute, cause,* or *effect*, rather than being based on something in the world, in terms of Jakobson and Lodge's typology. Understanding how both metaphor and metonymy can be used in design can illustrate how design patterns work alongside more traditional object-oriented modelling to produce designs that are accurate, flexible, and better understood.

Keywords: Object-Orientation, Design, Metaphor, Metonymy.

1 Introduction

Underlying most writing on object-oriented analysis and design is a single, simple idea: that objects in a program can model objects in the real world. To model a farm, for example, a program could have a Bovine class, where one object represented every cow; an Ovine class where every object represented a sheep; a Porcine class where every object represented a pig, and so on. Because, in the real world, pigs, sheep, and cattle are all kinds of farm animals, the Bovine, Ovine, and Porcine classes could all inherit from an abstract DomesticatedAnimal class. Because, again, "in the real world" (or down on the farm) the pigs, sheep, and cattle live in fields, you could have a GrazingArea class, one instance per field, and associate each DomesticatedAnimal with one GrazingArea, and so on. Comparing Figure 1a and Figure 1b, we can see that a UML instance diagram shows how a farm system can be seen as a model for the real world.

In this paper, we analyse the underlying rationale of object-oriented design. We begin by considering briefly the notion of the real world — the idea of a reality outside the program that it can model — and then go on to consider how the program is connected to this external reality. In Sections 3 and 4 we argue that most object-oriented design, and some of the most common design patterns, are *metaphorical*, that is, objects in the program are in some way like the objects in the real world to which they correspond. Then, in section 5 we investigate designs, particularly the advanced designs produced by design patterns, that are **not** metaphor; rather they are metonymy, that is, based on some cause, effect, or attribute of objects rather than the objects themselves — this distinction between metaphor and metonymy is based on Roman Jakobson and David Lodge's typology of literature [22, 17]. Finally, in section 6 we take a small detour to consider a few design patterns that are neither metaphor or metonymy, section 7 discusses related work, and then section 8 concludes the paper.

2 The Real World

> *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 [21]

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. This principle is often elevated to the a stronger claim that object-oriented is the "natural" way to program or to design: objects, like exotic fruit, are just there for the plucking [37, 20, 3]. More fundamentally, this quotation assumes that there is some kind of external "reality" (one of the few words that means nothing without quotes) [27] to which the program can refer. This is problematic for a number of reasons: not least that many programs (such as computer games with dragons and demons) have no reference to objective reality — according to this definition, at least, we can conceive of such programs modelling an "imagined reality" that does not exist, but still constitutes an external referent for the program.

Alternatively, rather than modelling an imaginary world, perhaps we *create* that world in the process of modelling it. There are no dragons, demons, or SMS Messages, even imaginary ones, until they are modelled in the process of systems development, and the process of systems development calls them into being. Furthermore, if this is the case for imaginary worlds, or technologies that do not exist prior to our software, perhaps this is true (at least to some extent) even when we implement software to computerise an existing manual business process or replace an existing software. As a mundane example, the software that runs the university where we work has pages of codes that describe students' enrolment status. These codes did not exist before the construction of the system: but neither did the precise distinctions of students statues that the codes represent. In other words, even if we model reality, we may not model a reality that preexists our software development effort.

Even where some preexisting reality is evident (say for a library system) defining that reality can be very difficult — what is a "book"? does this include an encyclopedia? How about a DVD? Even if reality can

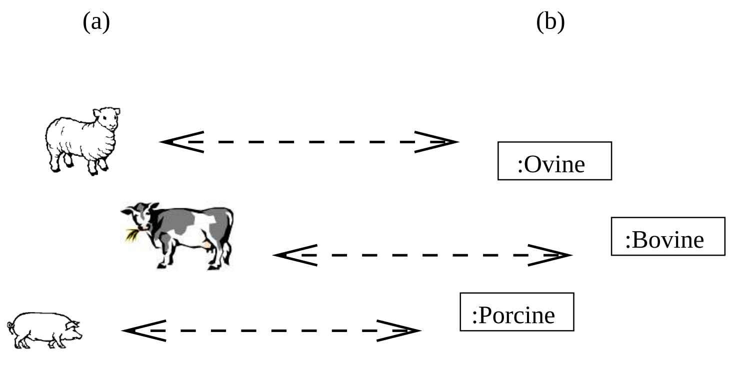

Figure 1: A Picture of farm animals, and the corresponding UML model

be well defined, the program's simulation of it may be tenuous at best [16] — for example, how can a library system know about untraced books that have been stolen from the library: the very idea of an "untraced stolen book" is something that *does not and cannot* be simulated in a computer system.

Also implicit in this quotation is the idea that the real world could be composed of objects, for otherwise, how could a program made of objects model reality? [20] The idea of a world made of objects underlies Western philosophy: Newtonian mechanics, for example, can be understood as describing a universe of abstract, encapsulated objects (abstract and encapsulated because an object's mass is modelled only by its centre of gravity) with internal states (velocities) that can be changed. A common fallacy expects the "objects" in the world to be the same kind of objects as the objects used in object-oriented programming, or even in a particular programming language (see for example [20, 3]) . This raises many questions: Do all objects in the real world have encapsulation? Do they have interfaces? Polymorphism? Constructors? What about "objects" in the real world such as "sunshine", "credit ratings", or "gravity" which do not seem to be the same kinds of objects as "cows" or "sheep".

To avoid these problems, in this paper we will follow the logical positivist tradition behind much computer science and information systems analysis, and assume that there is an external reality to which the program must relate, and that this "reality" can be analysed in terms of classes describing individual objects. We make these assumptions without making any claims that the external reality actually exists (it could be imaginary or contradictory) or that it is actually "object-oriented".

The question addressed by this paper is: ***how can we characterise the relationship between the classes and objects in the program and the classes and objects in the (real or imaginary) external world?***

3 Metaphoric Design

In the introduction, we have already presented an example of the most basic kind of object-oriented design: modelling a farm with a number of objects that represent what we choose to see as objects in the real farm. Thus, the cows, sheep, pigs, and so on of the real farm are modelled by the Bovine, Ovine, and Porcine classes in the object-oriented program. It is important to realise what "modelled by" here means: clearly it does **not** mean the the Bovine objects in the program physically eat grass, produce cowpats into which one can step, and contribute large volumes of methane and other gases to warming the biosphere.

signifier	**referent**	**signified**
"lion"	person	brave person
"lamb"	person	docile person
Bovine object	cow	cow-object
Ovine object	sheep	sheep-object

Figure 2: Metaphor

The traditional way to describe this relationship is to say that the objects in the program are "abstractions" of the real objects. As computer scientists, we are familiar with abstractions: for example, a stack is an abstraction that might be implemented by an array, a pointer, and some executable code; the stack is an abstraction because it elides many of the details of actual implementation. Unfortunately, it doesn't seem to make sense to say that a Bovine object in the program is an "abstraction" of a real cow in this way: it doesn't make sense to say that the object in the program is "implemented" by a cow in reality, or that the objects in the program are special kinds of cows which do not eat, excrete, or expire. Alternatively, following Plato, we could have an abstraction of a cow as the "ideal, immutable, eternal form" of a cow, perhaps corresponding to a Cow *class*, but, again, this kind of abstraction is not a good description of the relationship between the cow object and the real cow [33].

In this paper, we propose a different description of the relationship between programmatic object and external object: that this kind of relationship can be seen as *metaphor*. Metaphor is a figure of speech where one thing is described using terms that do not really apply to it. The word "metaphor" is derived from a Greek word for "transfer", so the effect of a metaphor is to transfer meaning from one thing to another. Cultural criticism has found metaphor an effective term for analysing many types of creative endeavour, including cinema, televising, advertising, multimedia, and popular music; Jakobson (and Lodge after him) has argued Metaphor is one the underlying structures of literature [22, 17, 8].

The first rows in Figure 2 shows how metaphor functions in speech. In phrases such as "he's a lion!" or "she's a lamb!" we use words (signifiers) "lion" or "lamb" as metaphors for a person to signify that that person is brave or docile.

This kind of relationship is precisely what we find in object-oriented modelling: an object in the program is a metaphor for an object in reality, and part of the meaning of the object in reality is transferred to the object in the program. The later rows of Figure 2 show how an object in a program can be a signifier

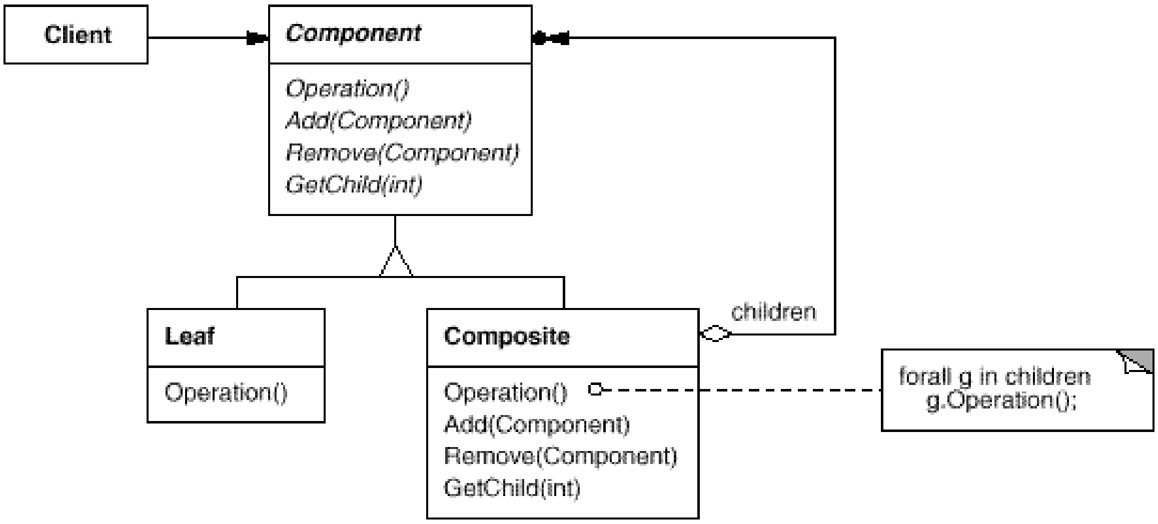

Figure 3: The structure of the Composite pattern [12]

for some referent in the world, signifying the object in the program models the object in the world . Thus, while our Bovine objects do not eat grass, they may have identity number, age, and weight attributes that model some features of the corresponding real cow.

We call this kind of design *Direct Metaphorical Design*: metaphorical because it is based on metaphor, and direct because the metaphors are being represented directly, using the constructs from most object-oriented programming languages or modelling notations. Direct metaphorical design is the most common and most basic kind of object-oriented design: for example, an accounting system may have one object per account; a game one object per demon and one per dragon, a university administration system one object per course and one per student.

Where direct metaphorical designs work, they are easy to produce, implement, understand, and modify: one simply identities the objects of interest in the real world, and creates corresponding objects in the design model.

4 Design Patterns and Indirect Metaphorical Design

Not all metaphorical designs can be produced directly. In other words, there are several kinds of objects (or, more typically, structures of objects) that are common in the worlds we wish to model, but that cannot be translated directly into object-oriented designs — perhaps due to weaknesses or non-orthogonality in programming or modelling languages [7, 11].

To give a couple of simple examples: modelling languages such as UML support multi-directional, multi-place relationships between objects that can can be used metaphorically (and directly) to model relationships 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 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 multi-way relationships, so this relationship needs to be encoded somehow into the program, typically by splitting the relationship between multiple objects, introducing extra objects to model the relationship, or both [28].

To consider another example, objects in the real world are often made up recursively from parts and wholes. Large organisations are made up of smaller organisational units; these units are composed of smaller units, in turn composed of still smaller ones. Quotations, bills of sale, insurance policies, and document contents all have a similar structure: a whole quotation is made up of a number of subquotations, each of which may have its own subquotations and so on. While some programming languages (such as Garnet, Amulet, and Keykit [26, 34]) support this kind of composite object, most programming languages and modelling notations do not, so once again, these structures need to be encoded using the features of the available languages and notations. Generally, these kind of recursive structures are modelling using the Composite [12] or Part-Whole [4] patterns (see Figure 3).

A design pattern is a "*description of communicating objects and classes that are customised to solve a general design problem in a particular context*" [12, p.3]. Designers can incorporate patterns into their program to address general problems in the structure of their programs' designs, in a similar way that algorithms or data structures are incorporated into programs to solve particular computational or storage problems. A growing body of literature catalogues patterns for object-oriented design, including reference texts such as *Design Patterns* book [12] or *Patterns of Software Architecture* [4], and patterns compendia such as the *Pattern Languages of Program Design* series [6, 36, 23].

These patterns are often counterintuitive to novice designers and programmers — although experienced programmers may find them quite obvious. Figure 3 shows the composite pattern from *Design Patterns*; this design exhibits a few rather strange features. First, classes representing the whole composition ("Composite" for example) are subclasses of classes representing the part ("Component"). That is, the most important class in this structure is the class representing the part, not the whole. Second, the crucial recursive relationship appears as a many-to-one aggregation from a subclass "Composite" to a superclass "Composite". To novices, this link appears backwards, going "up" the tree rather than down, from whole to parts, and the aggregation within an inheritance hierarchy appears completely arbitrary.

In spite of these caveats, the *result* of applying the Composite pattern (or a pattern for relationship, or whatever) is ultimately straightforward. Although the class diagram (Figure 3) for composite looks strange, Figure 4 shows the structure of the *objects* that the class diagram generates: the recursive structure in the real world we hoped to model.

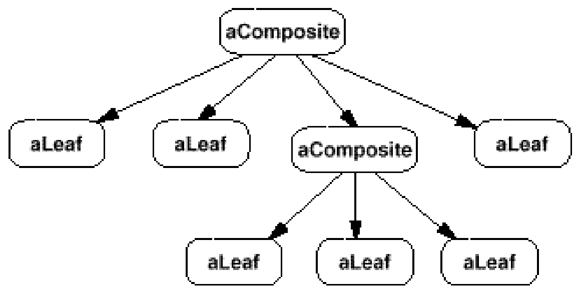

Figure 4: The objects created by the Composite pattern [12]

We call these kind of designs *indirect metaphorical designs* — they are metaphorical because the relationship between the resulting objects in the program and the objects in the world is metaphorical for objects in the world, and they are *indirect* because the program or modelling language (class) structures may not be obvious.

Other common examples of indirect metaphorical designs can be found in the Singleton pattern — where a class has only one instance; the Prototype pattern — where new objects are created by copying existing objects; and double dispatch — where the method that is selected depends upon the class of more than one of its arguments [12].

The difference between direct and indirect metaphorical designs lies in the features of the languages used to express them: some languages lack sufficient features to express required metaphors directly, so they must be encoded indirectly, such as by a design pattern. Translating between languages can change an indirect metaphoric design into a direct metaphoric design, and vice versa. For example, Java programs often require Prototype, Singleton and double dispatch to represent real-world features that could be encoded directly in languages such as CLOS or Cecil [19, 5].

5 Metonymic Design

Some — indeed many — design patterns don't make sense considered as metaphors. Consider State, one of the simpler patterns (see Figure 5). 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 an internal 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, we could use the State pattern to record the changing state of a sheep from newborn lamb, dipping, crutching, dagging, hogget, breeding ram, and finally to mutton.

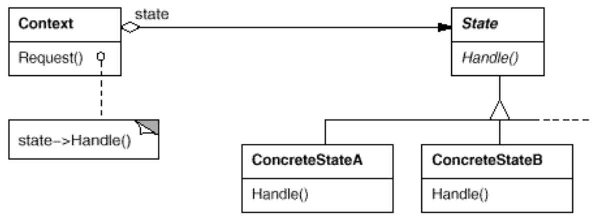

Figure 5: The structure of the State pattern [12]

Now, the implementation of the State pattern is quite straightforward, indeed, it is very direct: you add an extra object and class hierarchy to your 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 with dynamic classification or inheritance will reduce the amount of delegation code required in the context object, the overall geometry of the solution remains the same [35, 32, 10].

The State pattern raises an important question regarding the design or analysis of the program, however: *What object in reality does the state object represent?* In the farm example, the state object certainly doesn't model a subordinate physical object that is attached to a sheep and that changes throughout a 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 briefly another common pattern, the Iterator pattern (see Figure 6). The Iterator pattern is used to traverse through an 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. The Iterator pattern raises the same question for object-oriented design and analysis as the State pattern, however: *What object in reality does the iterator represent?* Unlike the State pattern, the Iterator must be part of the collection's public interface, and cannot be dismissed as a mere implementation detail.

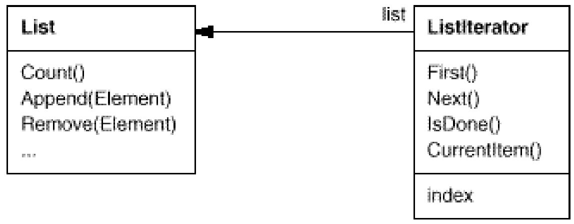

Figure 6: The structure of the Iterator pattern [12]

Our answer to this questions that that State and Iterator objects, and the similar objects introduced by many other patterns, do *not* represent objects from the real world: that is, they are *not* metaphorical, either directly or indirectly. Rather, these objects and these design patterns are exemplars of *metonymic* de-

signifier	referent	signified
"crown"	person	king
"law"	person	police officer
SheepState	sheep	state of the sheep
SheepIterator	flock of sleep	iteration cursor

Figure 7: Metonymy

signs, rather than metaphorical designs. According to Roman Jakobson and David Lodge, metaphor and metonymy are fundamental structures of language that transfer meaning, however they make different kinds of transfers.

To quote Robin Penrose:

> Metaphor is a figure of speech based on similarly, whereas metonymy is based on contiguity. In metaphor you substitute something *like* the thing you mean for the thing itself, whereas in metonymy you substitute some attribute or cause or effect of the thing for the thing itself.

Nice Work, David Lodge, Martin Secker & Warburg, London 1989.

Thus, calling King Richard the "lion-heart" is a metaphor — he is a king, not a lion — but calling him the "crown" is metonymy — wearing a crown is one attribute of being a king.

Figure 7 shows that metonymy in object-oriented software design functions in the same way as in language generally: comparing this with Figure 2 illustrates the differences and similarities between metonymy and metaphor. Again, the first two rows of Figure 7 are figures of speech: the signifiers "crown" or "law" applied to a person signify "king" and "police officer" respectively. The later rows of the table show examples from software.

The states of the sheep, for example, are metaphors for "real" objects own right, rather, they signify attributes of sheep. Similarly, the position of an iteration is not part of reality, rather it is a consequent effect of iterating through the collection.

Thus, the majority of the design patterns are metonymy, rather than metaphor. 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 requests, responsibilities, algorithms, interactions, snapshots of internal states, and traversal operations, respectively.

One intuitive way to determine whether a pattern (or a design) is metaphor or metonymy is to ask how hard the pattern is to explain: easy patterns that involve just one object tend to be metaphor (this is a composite object; this is a prototype which can be cloned) while the complex patterns involving multiple objects tend to be metonymy (this is part of the internal state of another object; this is a traversal position in a collection).

6 Programmatic Patterns

There are some patterns that are neither metaphor or metonymy. Consider the Facade pattern, for example (see 8). The Facade pattern inserts an extra interface into a program to encapsulate a set of objects forming a subsystem. This extra interface is typically nothing to do with any external reality, rather, it is purely about the internal structure of the software.

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. Again, this pattern is not about any kind of modelling or analysis of any kind of reality: rather is it purely about the internal structure of system.

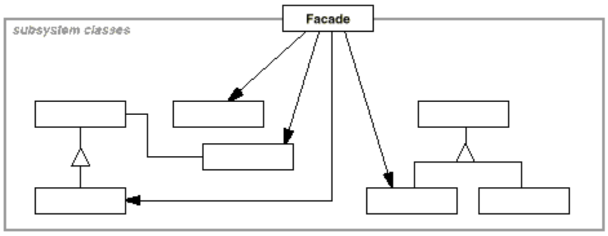

Figure 8: The Facade pattern [12]

We call these kind of patterns *programmatic*, because they are about programs' internal structure rather than their relation to an external reality. Many of the patterns in the book Patterns of Software Architecture are programmatic patterns, including the Layers, Blackboard, Pipes and Filters, and Counted Pointer patterns.

Note that however designs or patterns actually function, the *names* of patterns are almost always metaphorical. For example, the name of the Bridge pattern is a metaphor for the *class diagram* of the pattern — the diagram (see Figure 9) is supposed to look like a bridge between the "Abstraction" on the left and the "Implementation" on the right. In spite of this metaphoric name, the bridge pattern, focusing on separating abstractions and implementations, is strictly programmatic.

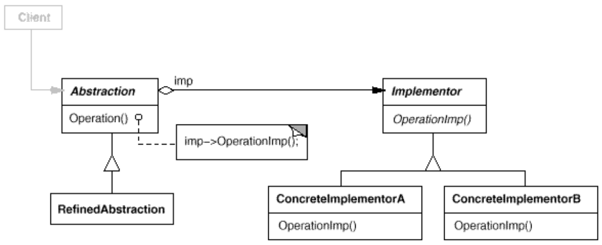

Figure 9: The structure of the Bridge pattern [12]

The final pattern we will consider in depth is the Strategy pattern. The structure of the pattern is basically the same as the State pattern we have discussed above: a context object delegates requests to an internal, replacable strategy object implements an algorithm, so that changing the strategy object changes the algorithm executed by the context (see Figure 10).

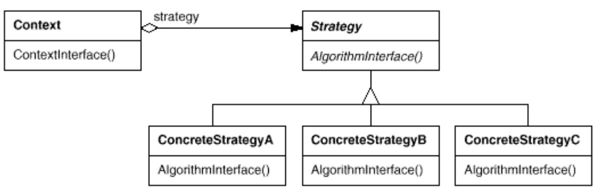

Figure 10: The structure of the Strategy pattern [12]

We consider the strategy pattern to be metonymy: the strategy object reifies an effect of its context. In some applications, however, the pattern can be used

Creational Patterns		
Abstract Factory	creates related objects	metonymy
Builder	creates complex objects	metonymy
Factory Method	creates subclasses	metonymy
Prototype	exemplary object	metaphor
Singleton	single instance	metaphor
Structural Patterns		
Adapter	converts interfaces	metaphor
Bridge	decouple abstraction and implementation	programmatic
Composite	model recursive tree structure	metaphor
Decorator	add responsibilities to objects	metonymy
Facade	interface for a subsystem	programmatic
Flyweight	save memory of similar objects	programmatic
Proxy	surrogate for access control	metaphor
Behavioural Patterns		
Chain of Responsibility	handle requests	metonymy
Command	request as an object	metonymy
Interpreter	interpret a language	programmatic
Iterator	iteration cursor	metonymy
Mediator	encapsulate interactions	metonymy
Memento	snapshot of objects' state	metaphor
Observer	update dependents	metonymy
State	change behaviour	metonymy
Strategy	vary algorithms	metonymy
Template Method	subclasses change algorithms	programmatic
Visitor	represent traversal operations	metonymy

Figure 11: Classification of Patterns

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 metaphorical. Furthermore the pattern can even be used programmatically, as Strategies can be used to avoid multiple inheritance, reducing the number of classes in the program. Ultimately, as shown in Figure 11, although patterns fall mostly in to one of the three categories — metaphor, metonymy, programmatic; and Strategy is primarily metonymy — a particular application of a pattern can fall into any category.

7 Related Work

Many texts present object-orientation design as fundamentally metaphorical, ranging for books aimed at novices [20], practitioners [30, 14, 15, 21] to theorists such as Abadi and Cardelli [1]. Responsibility Driven Design, for example, advises developers to model physical objects and conceptual entities (metaphor) but *not* attributes of objects (metonymy) [38]. 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 [24, 25].

Given their wide practical acceptance, there has been surprisingly little published on the underlying dynamics of design patterns, rather than accounts of patterns themselves. *Design Patterns* itself categorises patterns into three types, creational patterns are about creating objects, structural patterns about program structure, and behavioural patterns about program behaviour. This categorisation seems orthogonal to our classification of patterns according to metaphor and metonymy (see Figure 11 for the full classification). Egerbo and Cornils' have classified of patterns into those adequately described by other patterns; those described by existing language features; and those that are truly "fundamental" patterns [2]. The distinction between language dependent and fundamental patterns is similar to our distinction between direct and indirect patterns. Gill and Lorenz have similarly analysed patterns based on their degree of support in programming languages [13].

Metaphors are, of course, very common in user interface design — so much so we hardly perceive a Windows or Macintosh "desktop" as a metaphor. While metaphors are have been the subject of much study in human-computer interaction, this topic is only tangentially related to their user in the internal design of a software system [18, 29, 9]. Probably the most relevant work is Randy Smith's analysis of the tension between "literalism" and "magic" in the Alternate Reality Kit: literal objects represent physical objects (metaphor) while magical objects represent physical forces, causes and effects (metonymy) acting upon those objects [31].

8 Conclusion

In this paper, we have shown how features of object-oriented design patterns can be illuminated by Jakobson and Lodge's theory of metaphor and metonymy — the full classification is shown in Figure 11. Object-oriented design is primarily metaphorical, however, metaphorical designs that cannot be implemented directly in a programming or design language give rise to patterns (such as Composite and Prototype) corresponding to indirect implementations of these metaphorical designs. Modelling attributes, causes, and effects (rather than real world objects) produces metonymic designs and more advanced patterns. Finally, designs that improve the internal structure of the program without reference to an external reality give rise to programmatic patterns.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1996.

[2] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In *OOPSLA Proceedings*, pages 134–143. ACM, 1998.

[3] David William Brown. *An Introduction to OBJECT-ORIENTED ANALYSIS: Objects and UML in Plain English.* John Wiley & Sons, 2002.

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

[5] Craig Chambers. The Cecil language: Specificaion and rationale. Technical report, Department of Computer Science and Engineering, The University of Washington, 1997.

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

[7] 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.

[8] Umberto Eco. *A Theory of Semiotics.* Indiana University Press, 1976.

[9] Thomas D. Erickson. Working with interface metaphors. In Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton, and Saul Greenberg, editors, *Readings in Human-Computer Interaction*, chapter 2, pages 147–151. Morgan-Kaufmann, second edition, 1995.

[10] Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP Proceedings*, pages 186–211, Brussels, July 1998. Springer-Verlag.

[11] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community.* Oxford University Press, 1996.

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

[13] Joseph (Yossi) Gil and David H. Lorenz. Object technology: Design patterns and language design. *IEEE Computer*, 31(3):118–120, March 1998.

[14] Brian Henderson-Sellers. *A BOOK of Object-Oriented Knowledge.* Prentice-Hall, 1994.

[15] Brian Henderson-Sellers and Julian M. Edwards. *BOOKTWO of Object-Oriented Knowledge: The Working Object.* Prentice-Hall, 1994.

[16] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley, 2001.

[17] Roman Jakobson. Two aspects of language and two types of linguistic disturbance. In Roman Jakobson and Morris Halle, editors, *Fundamentals of Language*, page 58. Mouton, Den Haag, 1956.

[18] J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. Irby, M. Beard, and K. Mackey. The Xerox Star: A retrospective. *IEEE Computer*, 22(9), 1989.

[19] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, 1989.

[20] Gene Korienek and Tom Wrensch. *A Quick Trip To Objectland.* Prentice-Hall, 1991.

[21] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

[22] David Lodge. *The Modes of Modern Writing.* Edward Arnold, 1977.

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

[24] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[25] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, second edition, 1997.

[26] Brad A. Myers, Rich McDaniel, Rob Miller, Brad Vander Zanden, Dario Guise, David Kosbie, and Andrew Mickish. The prototype-instance object systems in Amulet and Garnet. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 7. Springer-Verlag, 1999.

[27] Vladimir Nabokov. *Lolita.* Putnam, New York, 1958.

[28] James Noble. Basic relationship patterns. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, chapter 6, pages 73–94. Addison-Wesley, 2000.

[29] Jenny Preece, editor. *Human Computer Interaction.* Addison-Wesley, 1994.

[30] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, New Jersey, 1991.

[31] Randall B. Smith. The alternate reality kit: an example of the tension between literalism and magic. *Computer Graphics and Applications*, 7(9), 1987.

[32] Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, 6(3):25–32, June 1993.

[33] Antero Taivalsaari. Classes vs. prototypes: Some philosophical and historical observations. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 1. Springer-Verlag, 1999.

[34] Tim Thompson. Keynote — a language and extensible graphics editor for music. *Computing Systems*, 3(2):331–357, 1990.

[35] David Ungar and Randall B. Smith. SELF: the Power of Simplicity. *Lisp And Symbolic Computation*, 4(3), June 1991.

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

[37] Oscar Wilde. *The Importance of being Earnest: a trivial comedy for serious people.* Leonard Smithers, 1988.

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