

Object Relations and Syntactic Mechanisms in Design Patterns

URIEL COHEN

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

urielc@cs.technion.ac.il

Abstract

This work investigates the syntactical mechanisms used in design patterns. A taxonomy of patterns is presented, and a data set of design patterns is selected from this taxonomy. We argue that composition is the syntactical mechanism most used in design patterns. Several criteria to classify and implement composition in our data set of design patterns are suggested. We postulate that the knowledge of design patterns should be incorporated into object-oriented languages using the analyzed syntactical mechanisms.

1 Introduction

One of the main problems of software engineering is to capture the knowledge of expert software designers and engineers in order to allow other less experienced practitioners to apply that knowledge in improving their designs. *Design patterns* were invented precisely for this purpose: to explain and record a recurring design in object-oriented programming [9].

It has been already demonstrated that design patterns succeed in capturing design knowledge. We think that the time has come to take this knowledge and incorporate it into modern object-oriented languages. We are particularly interested in one mechanism used repetitively throughout the design patterns: composition.

Although design patterns are, in theory, independent of the language in which they are implemented, in practice this is not always true. Different programming languages within the object-oriented paradigm provide different features, and these dictate what can and cannot be implemented easily in the language. On the one hand, a design pattern is not very worth if it can be implemented as the instantiation of one language feature. This is sometimes called an *idiom* [7]. On the other hand, a design pattern may require a complex relationship between several classes and objects. Gil and Lorenz give a taxonomy of design patterns based on this criteria [10].

Design patterns have several drawbacks when implemented in languages like SMALLTALK [11], C++ [14] and JAVA [2]. These problems can be summarized as follows:

- *Traceability problem.* In many cases, the design pattern is lost between the classes and objects that implement it, making it hard to see the whole pattern as one entity. When this is the case, a good documentation tool is required, even with a reference to the book or article in which the implemented pattern is classified.
- *Reusability problem.* A design pattern is usually implemented as the relation between several objects or classes, which pass messages between them. Since these objects and classes have other functions in the system, it is difficult to reuse the pattern implementation in some other application without the need to make a thorough modification.
- *Implementation overhead.* An engineer is required to implement again and again, in every project, a set of very simple or trivial methods that are needed for the specific pattern.

These problems have been described in detail by Bosch [4].

The motivation of this research is the belief that incorporating design patterns into object-oriented languages like C++ or JAVA as language constructs will make an advance in the problems here presented.

In this work we investigate the syntactic mechanisms that allow the implementation of design patterns and the relations between the objects that compose them, in order to reach a better understanding of the set of lingual features that would permit us to easily implement them. This will lead to a new classification of design patterns according to criteria based on syntactics rather than semantics (i.e., the mechanism of implementation rather than the purpose of the design pattern).

The rest of this paper is organized in the following way. In the next section we present a taxonomy of design patterns and select a data set to base our work on. Section 3 discusses the use of composition in design patterns, and Section 4 presents several criteria used to implement it. Finally, our conclusions are offered in Section 5.

Domain specific		Concurrency	17
		Distribution	9
		Virtual machines	2
		<i>Total</i>	28
General purpose	Sentence level	Clichés	4
		Idioms	3
		<i>Total</i>	7
	Module level	Procedural patterns	10
		Design patterns	29
		Compound patterns	3
		Variations	12
	<i>Total</i>	54	
	Architectural level	Architectural patterns	6
		<i>Total</i>	6
<i>Total in taxonomy</i>			95

Table 1: A taxonomy of patterns & n^o of patterns in each category.

2 Design Patterns Data Set

Gamma, Helm, Johnson and Vlissides published in 1995 the first design patterns catalogue [9]. Since then, the increasing numbers and popularity of design patterns lead several other authors to make their attempt at compiling a definitive catalogue of the field, or at least a system for the classification of patterns. Buschmann et al. [5] catalogue uses the abstraction level as the principal classification criterion, distinguishing between idioms, “plain” design patterns, and architectural patterns. Tichy’s catalogue [15], which comprises over a hundred design patterns, classifies the patterns by their application domain.

We also made an attempt at classifying the design patterns into categories. We do not intended to compile a definitive catalogue of patterns. Our sole purpose was to compile a taxonomy detailed enough to be able to select a data set of design patterns on which our study will be based on. We based our taxonomy on Tichy’s work [15], which includes both the Gang of Four (GoF) [9] and Buschmann et al. patterns. The taxonomy is presented on Table 1, which also includes the number of patterns in each category.

Ideally, we would like our data set to include every catalogued pattern. In practice, this is not convenient or even possible because of several reasons: some patterns are too simple to be considered a design pattern, some do not apply to object-oriented languages, some are restricted to a very specific domain, others can be decomposed into a collection of simpler design patterns, etc. In summary, the main category in our taxonomy that fits the purposes of our research is the one called *design patterns*.

The complete list of patterns in our data set is (29 patterns): *Abstract Factory*, *Adapter*, *Bridge*, *Builder*, *Chain of Responsibility*, *Command*, *Composite*, *Decorator*, *Extension Objects* [8], *Facade*, *Factory Method*, *Flyweight*, *Interpreter*, *Iterator*, *Manager* [13], *Mediator*, *Memento*, *Observer*, *Pipeline* [5, 17], *Product Trader* [3], *Prototype*, *Proxy*, *Sponsor-Selector* [18], *State*, *Strategy*, *Template Method*, *Type Object* [12], *Visitor* (patterns for which no reference is given are from GoF [9]).

3 Composition as Pseudo Inheritance

Composition is a syntactic technique that allows the reuse of code in a black-box fashion, without needing to expose the implementation details of the component being reused. To apply this mechanism, an object includes a reference pointing to the reusable component; alternatively, the object can define a data member that is a real instance of the component to reuse. Both *inheritance* and *object composition* are recurring techniques in software design for reuse of code and components. Although inheritance¹ is a feature present in every object-oriented programming language, design patterns rely mostly on composition. In this work we intend to find out how composition is used in design patterns. Nevertheless, inheritance is still necessary in design patterns. In fact, object composition and inheritance should be used in combination to achieve more flexible designs that exploit existing components to their maximum.

Table 2 presents a classification of design patterns according to the main language feature used to implement it. We refer to this main feature in terms of inheritance. As seen in the table, most design patterns lie under the classification of *pseudo inheritance*. We can also see that with the exception of SINGLETON and MEMENTO, every design pattern in our data set uses inheritance in one way or another.

¹In this work, we make a distinction between *inheritance*, used only when inheriting from a non-abstract class with code reuse intention, *abstract classes*, a base class with some of its methods abstract, and *interfaces*, a base class with all of its methods abstract.

	Purpose		
	Creational	Structural	Behavioral
Inheritance		Adapter (class)	
Abstract base class	Factory Method		Template Method
Pseudo inheritance (Composition)	Abstract Factory* Builder* Product Trader* Prototype*	Adapter (object)* Bridge* Composite* Decorator*(i) Extension Objects* Facade Flyweight* Manager* Proxy* Type Object(i)	Chain of Responsibility(i) Command* Interpreter* Iterator* Mediator* Memento Observer* Pipeline*(i) Sponsor-Selector* State* Strategy* Type Object(i) Visitor*
No inheritance	Singleton		

*Design patterns using interfaces.

(i)This pattern also includes an abstract base class, but the main language feature used is still pseudo inheritance.

Table 2: Classification of design patterns according to the main language feature used.

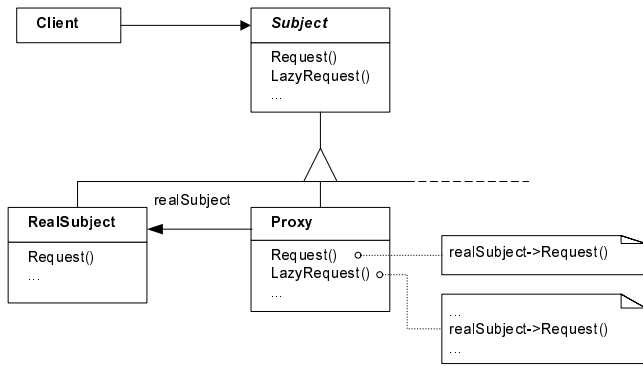


Figure 1: The structure of the PROXY pattern.

4 Implementing Composition

In this section we will analyze different syntactic mechanisms and criteria used to implement pseudo inheritance: *coupling strength*, *time of coupling*, *internal dynamic binding*, *features export*, *overriding prior to export*, and *external dynamic binding*. This criteria set was first listed by Gil and Lorenz [10], but never applied in practice. To illustrate the discussion we will use the PROXY pattern as our running example, depicted in Figure 1. As an illustrating example of these criteria we propose the PROXY pattern.

1. *Coupling strength*. This criteria stands to answer the question “how strong is the connection between the participants of the composition?”. The term *coupling* refers to the action of pairing the objects together. Depending on the strength of the bond between the participants, the component in the composition, perhaps, could be replaced. This is the case of a syntactic implementation of the composition which uses *object reference semantics*. Such an implementation is a much looser connection than *coupling by value*, and the referenced objects may be changed at runtime, which is beneficial to a flexible design and thus used throughout most of the design patterns.

The PROXY pattern can only be implemented using reference semantics. Moreover, it requires pointer semantics, i.e., the references used in the implementation may accept a NULL value before their initialization. The components in this pattern (RealSubject) are single objects that can be shared among other compositions (i.e., referenced from other objects also).

2. *Time of coupling*. Time of coupling is indicative of the time in which the objects get paired together, i.e., “when is the connection established?” It is directly related to the coupling strength of the composition: if the connection between the objects (the owner and the component) uses value, reference or pointer semantics, then it will be established at a different time. The time of coupling is very closely associated with the time of construction of an object. A component cannot be connected to the owner before the component is created. Even so, it is relevant to ask “when is the component created?”, and “who is responsible for its creation?”

The PROXY pattern uses a lazy creation time of coupling. In this case, the owner (Proxy) creates and attaches the component (RealSubject) after he himself is created, but once it is done, both objects have the same life span. RealSubject is deleted with the destruction of Proxy and, thus, detached from the composition.

3. *Internal dynamic binding*. The purpose of this mechanism is to divert a message sent to the owner, to the component, in order to take advantage of the component’s implementation. The owner calls a method of the component as a reaction to a message sent to him by some client. This syntactic mechanism is called *forwarding*. However, in order to make composition as powerful as inheritance, an explicit reference to the owner may be passed as a parameter of the request. This way, the component will get access to its owner from within the method. This variation is called *delegation*.

We have classified our data set of design patterns according to their internal dynamic binding: whether or not it is necessary or useful to implement them using delegation. Most of the design patterns do not need to use delegation for their implementation, the use of forwarding is sufficient.

The PROXY pattern finds necessary to use forwarding in its implementation, but the extra feature present in delegation is unnecessary.

4. *Features export*. In a composition, the owner usually uses the component’s services to implement its extra functionality. Sometimes this is not enough: one would like to offer access to the component’s methods directly, taking full advantage of code-reusability. Inheritance, on the other hand, has a built-in mechanism to achieve this same goal. Every object from a derived class contains a sub-object from the base class. Through this sub-object one can access the inherited member functions. This is called *exporting the features* of an object, and depends also on the level of encapsulation of the sub-objects. To be able to export features (methods) from a composition, some implementation is required: we will use *forwarding* in order to do it. Nevertheless, we will not require the same parameters from both of the methods, nor the same name. We will only require the same semantical meaning from both methods (the exported and the exporting).

The PROXY pattern has two different groups of methods that are exported: Request() and LazyRequest().

5. *Overriding prior to export*. In addition to being exported, a method from a composition may also be overridden before it is exported. There are two ways to override a method: it can be *refined* (adding some extra implementation), or it can be *replaced* (modifying the implementation). In the former case, the owner’s method will call the component’s method, doing some additional operations before or after the call. In the latter, the owner will not call the component’s method, but it will provide an alternative implementation.

The Request features are simple exports without any kind of overriding. They only forward the requests to RealSubject. On the other hand, LazyRequest methods are overridden prior to being exported. They are handled by Proxy (replacing the component's methods), and only forwarded to RealSubject in special cases, hence refining their implementation.

6. *External dynamic binding.* This criteria is known also as *substitutability* of the component in the composition. It intends to answer the question “is the connection in the composition constant?”. The external dynamic binding of the objects in the composition is very closely related to the coupling strength and the time of coupling of the composition. It can even be deduced from those mechanisms. If a composition is implemented with value semantics, then the connection between the objects will be constant during their entire life span. The same can be said if the connection is established at *creation* or *lazy creation* time of coupling. However, if the connection is established at a *dynamic* time of coupling, then will not be constant, and the component will probably be replaced.

Although the PROXY pattern uses object reference semantics in its implementation, the connection between Proxy and RealSubject is constant once the component is created and attached.

5 Conclusions

The idea of introducing design patterns as language constructs is not a new one. However, not much was done along these lines. It has been debated in the past by Chambers, Harrison and Vlissides [6]. It has also been implemented, in our opinion, unsuccessfully. Bosch [4] implemented an object layer model called LayOM, where each object was composed of a number of layers which filtered the messages the object received. In his approach, design patterns were implemented by composition of several of these layers. Since the layers themselves were not flexible and some of the patterns were still composed of several objects, it didn't solve the problems presented in Section 1.

A different approach was adopted by Agerbo and Cornils [1]. They implemented a library of classes composing the *Fundamental Design Patterns*, according to their taxonomy. Using a pattern in an application required inheriting from the library classes. Although this approach is better than the one by Bosch, it used language features not found in JAVA- or C++-like languages.

In this work, we have analyzed the syntactic mechanisms present in design patterns, in order to obtain a better understanding of the possible lingual features that can be introduced into languages like JAVA or C++. By investigating a large number of design patterns we can gain insight into the language constructs that could be useful and the ones that are frequently found in design patterns. We strongly believe that this should be the first step in the long road of shaping future mainstream languages.

References

- [1] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. In *Proceedings of the 13th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, Oct.18-22 1998. OOP-SLA'98, Acm SIGPLAN Notices 33(10) Oct. 1998.
- [2] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.
- [3] D. Bäumer and D. Riehle. Product trader. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
- [4] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.
- [5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, 1996.
- [6] C. Chambers, B. Harrison, and J. Vlissides. A debate on language and tool support for design patterns. In *27th Symposium on Principles of Programming Languages, POPL'00*, Boston, MA, Jan. 2000. ACM SIGPLAN — SIGACT, ACM Press.
- [7] J. Coplien. *Advanced C++ Programmings Styles and Idioms*. Addison-Wesley, 1992.
- [8] E. Gamma. The extension objects pattern. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.
- [10] J. Gil and D. H. Lorenz. Design patterns vs. language design. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag. Workshop on Language Support for Design Patterns and Frameworks.
- [11] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.
- [12] R. Johnson and B. Woolf. Type object. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
- [13] P. Sommerlad. Manager. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
- [14] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
- [15] W. F. Tichy. A catalogue of general-purpose software design patterns. In *Proceedings of the 23rd Technology of Object-Oriented Languages and Systems Conference*, pages 330–339, Aug. 1997.
- [16] D. Ungar and R. B. Smith. SELF: The power of simplicity. In N. K. Meyrowitz, editor, *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–241, Orlando, Florida, Oct. 4-8 1987. OOPSLA'87, Acm SIGPLAN Notices 22(12) Dec. 1987.
- [17] A. Vermeulen, G. Begeed-Dov, and P. Thompson. The pipeline design pattern. In *OOPSLA '95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, Oct. 1995.
- [18] E. Wallingford. Sponsor-selector. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.