Object-Oriented Design Patterns

Wolfgang Pree

Applied Computer Science
University of Constance, D-78457 Constance, Germany
Voice: +49-7531-88-44-33; Fax: +49-7531-88-35-77
E-mail: pree@acm.org

Abstract. There is an undeniable demand to capture already proven and matured object-oriented design so that building reusable object-oriented software does not always have to start from scratch. The term *design pattern* emerged as buzzword that is associated as a means to meet that goal. This paper starts with an overview of relevant design pattern approaches. It goes on to discuss the few essential design patterns of flexible object-oriented architectures, so-called frameworks. The paper sketches the relationship between these essential design patterns and the design pattern catalog by Erich Gamma et al. [8]. The implications for finding domain-specific design patterns are outlined.

Keywords. Design patterns, object-oriented design, object-oriented software development, frameworks, reusability

1 Introduction

Over the past couple of years (design) patterns have become a hot topic in the software engineering community. In general, patterns help to reduce complexity in many real-life situations. For example, in many situations the sequence of actions is crucial in order to accomplish a certain task. Instead of having to choose from an almost infinite number of possible combinations of actions, patterns allow the solution of problems in a certain context by providing time-tested combinations that work. What does this mean in the realm of software construction?

Programmers tend to create parts of a program by imitating, though not directly copying, parts of programs written by other, more advanced programmers. This imitation involves noticing the pattern of some other code and adapting it to the program at hand. Such imitation is as old as programming.

The design pattern concept can be viewed as an abstraction of this imitation activity. In other words, design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development. As a consequence, books on algorithms also fall into the category of general design patterns. For example, sorting algorithms describe how to sort elements in an efficient way depending on various contexts. If the idea sketched above is applied to object-oriented software systems, we speak of design patterns for object-oriented software development.

Currently, patterns comprise a wide variety of activities in the software development process, ranging from high-level organizational issues such as project management and team organization to low-level implementation issues such as tips and tricks regarding the use of a particular programming language. The pattern conference proceedings reflect this wide spectrum, for example, [6]. Due to the origin of the pattern movement in the framework community, numerous patterns still focus on how to construct frameworks. Section 3 discusses these approaches in detail.

2 History and Overview of Pattern Approaches

The roots of object-oriented design patterns go back to the late 1970s and early 1980s. The first available frameworks such as Smalltalk's Model-View-Controller (MVC) framework [13] and MacApp [18, 21] revealed that a framework's complexity is a burden for its (re)user. A framework user must become familiar with its design, that is, the design of the individual classes and the interaction between these classes, and maybe with basic object-oriented programming concepts and a specific programming language as well. This is why (framework) cookbooks have come to light:

Framework Cookbooks. Cookbooks contain numerous *recipes*. They describe in an informal way how to use a framework in order to solve specific problems. The term framework usage expresses that a programmer uses a framework as a basis for application development. A particular framework is adapted to specific needs. Recipes usually do not explain the internal design and implementation details of a framework. Cookbook recipes with their inherent references to other recipes lend themselves to presentation as hypertext.

Cookbooks exist for various frameworks. For example, Krasner and Pope present a cookbook for using the MVC framework [13]. The MacApp cookbook [5] describes how to adapt the GUI application framework MacApp in order to build applications for the Macintosh. Ralph Johnson wrote a cookbook [12] for the HotDraw framework, a system developed by Kent Beck and Ward Cunningham for implementing various kinds of graphic editors. ParcPlace-Digitalk Smalltalk provides an extensive cookbook for adapting the corresponding framework library.

Coding Styles & Idioms. These patterns form a quite different pattern category. C++ is a representative example of an object-oriented programming language whose complexity requires coding patterns to tame its montrosity. The principal goals of coding patterns are

- to demonstrate useful ways of combining basic language concepts
- to form the basis for standardizing source-code structure and names
- to avoid pitfalls and to weed out deficiencies of object-oriented programming languages, which is especially relevant in the realm of C++.

Coplien's C++ Styles & Idioms [7] and Taligent's guidelines for using C++ [19] fall into this category.

Formal Contracts. In the early 1990s more advanced design pattern approaches gained momentum that focus on framework development. Available object-oriented analysis and design methods appeared to be insufficient to construct reusable software architectures. In order to better understand object-oriented architectures, they should be described on an abstraction level that is higher than their implementation language. For example, Richard Helm *et al.* described interactions between objects of different classes in a framework in a formal way [11].

Gamma's Description of ET++. Pioneering work was accomplished by Erich Gamma in his doctoral thesis [9] which presents patterns incorporated in the GUI application framework ET++ [20, 1]. Gamma was inspired by Helm's formal contracts. Instead of using a formal notation he decribed ET++ by means of informal text combined with class and interaction diagrams. The design pattern catalog [8], also known as Gang-of-Four (GoF) book, resulted from Gamma's PhD thesis.

Influence of Building Architecture. At the 1991 Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Conference—a major forum for researchers and practitioners in the field of object-oriented technology—Bruce Anderson headed the workshop "Towards an Architecture Handbook". Participants were encouraged to describe design patterns in a manner similar to the descriptions of architecture patterns presented in Christopher Alexander's books *A Pattern Language* [4] and *The Timeless Way of Building* [3]. These books show non-architects what good designs of homes and communities look like. One pattern, for example, recommends placing windows on two sides of a room instead of having windows only on one side. Alexander's patterns cover different levels of detail, from the arrangement of roads and various buildings to the details of how to design rooms. In general, this architecture handbook workshop inspired the pattern community, in particular the writing of the GoF book.

Workshops on object-oriented patterns were organized at subsequent OOPSLA conferences. A separate conference on design patterns (PLoP; Pattern Languages of Program Design) started in the U.S. in August 1994, the European pendant, called EuroPLoP, was first held in 1995.

3 Essential Framework Design Patterns

Frameworks are well suited for domains where numerous similar applications are built from scratch again and again. A framework defines a *high-level language* with which applications within a domain are created through *specialization* (= adaptation). Specialization takes place at points of predefined refinement that we call *hot spots*. We consider a framework to have the quality attribute *well designed* if it provides adequate hot spots for adaptations. For example, Lewis et al. present various high-quality frameworks [14].

3.1 Flexibility Through Hooks

Methods in a class can be categorized into socalled hook and template methods: Hook methods can be viewed as place holders or flexible hot spots that are invoked by more complex methods. These complex methods are usually termed template methods [8, 15]. Note that template methods must not be confused with the C++ template construct, which has a completely different meaning. Template methods define abstract behavior or generic flow of control or the interaction between objects. The basic idea of hook methods is that overriding hooks through inheritance allows changes of an object's behavior without having to touch the source code of the corresponding class. Figure 1 exemplifies this concept which is tightly coupled to constructs in common object-oriented languages. Method t() of class A is the template method which invokes a hook method h(), as shown in Figure 1(a). The hook method is an abstract one and provides an empty default implementation. In Figure 1(b) the hook method is overridden in a subclass A1.

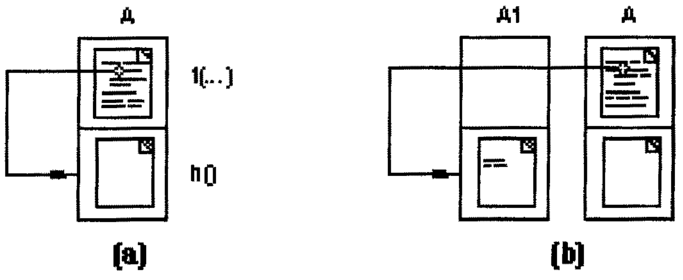

Fig. 1. (a) Template and hook methods and (b) hook overriding.

Let us define the class that contains the hook method under consideration as *hook class* H and the class that contains the template method as *template class* T. A hook class quasi parameterizes the template class. Note that this is a context-dependent distinction regardless of the complexity of these two kinds of classes. As a consequence, the essential set of flexibility construction principles can be derived from considering all possible combinations between these two kinds of classes. As template and hook classes can have any complexity, the construction principles discussed below scale up. So the domain-specific semantics of template and hook classes fade out to show the clear picture of how to achieve flexibility in frameworks.

3.2 Unification versus separation patterns

In case the template and hook classes are unified in one class, called TH in Figure 2(a), adaptations can only be done by inheritance. Thus adaptations require an application restart.

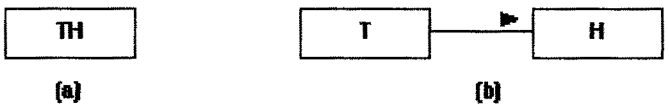

Fig. 2. (a) Unification and (b) separation of template and hook classes.

Separating template and hook classes is equal to (abstractly) coupling objects of these classes so that the behavior of a T object can be modified by composition, that is, by plugging in specific H objects.

The directed association between T and H expresses that a T object refers to an H object. Such an association becomes necessary as a T object has to send messages to the associated H object(s) in order to invoke the hook methods. Usually an instance variable in T maintains such a relation. Other possibilities are global variables or temporary relations by passing object references via method parameters. As the actual coupling between T and H objects is an irrelevant implementation detail, this issue is not discussed in further detail. The same is true for the semantics expressed by an association. For example, whether the object association indicates a *uses* or *is part of* relation depends on the specific context and need not be distinguished in the realm of these core construction principles.

A separation of template and hook classes also forms the precondition of *run-time adaptations*, that is, subclasses of H are defined, instantiated and plugged into T objects while an application is running. Gamma et al. [8] and Pree [16] discuss some useful examples.

3.3 Recursive combination patterns

The template class can also be a descendant of the hook class (see Figure 3(a)). In the degenerated version, template and hook classes are unified (see Figure 3(b)). The recursive compositions have in common that they allow building up directed graphs of interconnected objects. Furthermore, a certain structure of the template methods, which is typical for these compositions, guarantees the forwarding of messages in the object graphs.

The difference between the simple separation of template and hook classes and the more sophisticated recursive separation is that the playground of adaptations through composition is enlarged. Instead of simply plugging two objects together in a straightforward manner, whole directed graphs of objects can be composed. The implications are discussed in detail in [15, 16, 17].

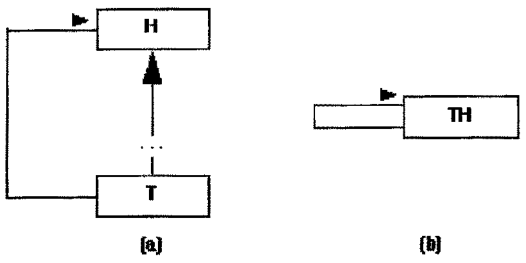

Fig. 3. Recursive combinations of template and hook classes.

3.4 Hooks as name designators of GoF pattern catalog entries

Below we assume that the reader is familiar with the patterns in the pioneering Gang-of-Four catalog [8]. Numerous entries in the GoF catalog represent small frameworks, that is, frameworks consisting of a few classes, that apply the essential construction patterns in various more or less domain-independent situations. So these catalog entries are helpful when designing frameworks, as they illustrate typical hook semantics. In general, the names of the catalog entries are closely related to the semantic aspects that are kept flexible by hooks.

Patterns based on template-hook separation. Many of the framework-centered catalog entries rely on a separation of template and hook classes (see Figure 1(b)). The catalog pattern Bridge describes this construction principle. The following catalog patterns are based on abstract coupling: Abstract Factory, Builder, Command, Interpreter, Observer, Prototype, State and Strategy. Note that the names of these catalog patterns correspond to the semantic aspect which is kept flexible in a particular pattern. This semantic aspect again is reflected in the name of the particular hook method or class. For example, in the Command pattern "when and how a request is fulfilled" [8] represents the hot spot semantics. The names of the hook method (Execute()) and hook class (Command) reflect this and determine the name of the overall pattern catalog entry.

Patterns based on recursive compositions. The catalog entries Composite (see Figure 3(a) with a 1: many relationship between T and H), Decorator (see Figure 3(a) with a 1:1 relationship between T and H) and Chain-of-Responsibility (see Figure 3(b)) correspond to the recursive template-hook combinations.

4 How to find domain-specific patterns

Hot spot identification in the early phases (eg, in the realm of requirements analysis) should become an explicit activity in the development process. There are two reasons for this: Design patterns, presented in a catalog-like form, mix construction principles and domain specific semantics as sketched above. Of course, it does not help much, to just split the semantics out of the design patterns and leave framework designers alone with bare-bone construction principles. Instead, these construction principles have to be combined with the semantics of the domain for which a framework has to be developed. Hot spot identification provides this information. Figure 4 outlines the synergy effect of essential construction principles paired with domain-specific hot spots. The result is design patterns tailored to the particular domain.

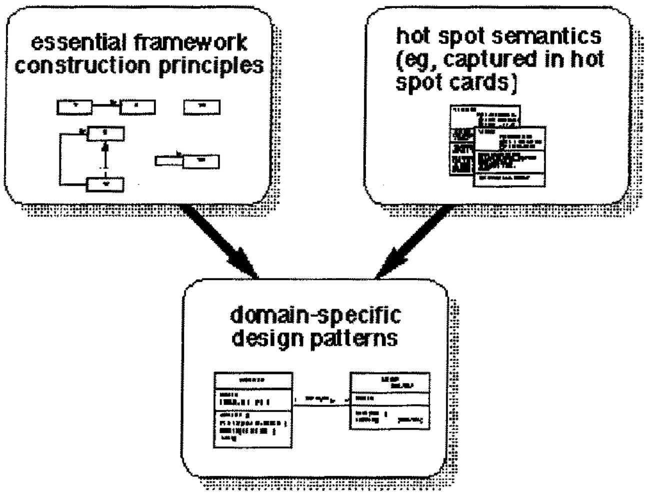
essential framework construction principles
hot spot semantics (eg, captured in hot spot cards)
domain-specific design patterns

Fig. 4. Essential construction principles + hot spots = domain-specific design patterns

Hot spot identification can be supported by hot spot cards, a communication vehicle between domain experts and software developers. Pree [16, 17] presents the concept of hot spot cards and detailed case studies where they are applied.

A further reason why explicit hot spot identification helps, can be derived from the following observations of influencing factors in real-world framework development: One seldom has two or more similar systems at hand that can be studied regarding their commonalities. Typically, one too specific system forms the basis of framework development. Furthermore, commonalities should by far outweigh the flexible aspects of a framework. If there are not significantly more standardized (= frozen) spots than hot spots in a framework, the core benefit of framework technology, that is, having a widely standardized architecture, diminishes. As a consequence, focusing on hot spots is likely to be more successful than trying to find commonalities.

5 Outlook

Are patterns just a hype? Lewis *et al.* [14] view the pattern movement from the perspective of frameworks as part of the evolution of this technology: "Patterns ... is one of the most recent fads to hit the framework camp. ... Expect more buzzwords to appear on the horizon." Because patterns have become the vogue in the software engineering community, the term is used now wherever possible, adorning even project management or organizational work. So the genericity of the term pattern might be the reason that patterns are found everywhere, a fact which is regarded as a clear indication of a hype.

Nevertheless, we view pattern catalogs, in particular the GoF-catalog, as important first step towards a more systematic construction of flexible object-oriented architectures. There is no doubt that organizational measures are at least equally important to be successful as framework development requires a radical departure from today's project culture. Goldberg and Rubin [10] discuss this in detail, without using the term pattern for these management issues.

Probably, we should read the recent books published by the building architect Christopher Alexander [2] in order to predict what will happen to software patterns. He states that the cataloging of architectural styles did not really help architects to come up with buildings that have what he calls a quality without a name. A reduction to very few principles, all related to the concept of "center", allow the generation of this quality without a name. Let us wait and see what this means in the realm of software.

6 References

1. Ackermann P. (1996). *Developing Object-Oriented Multimedia Software—Based on the MET++ Application Framework.* Heidelberg: dpunkt.Verlag
2. Alexander C. (1997). *The Nature of Order.* New York: Oxford University Press
3. Alexander C. (1979). *The Timeless Way of Building.* New York: Oxford University Press
4. Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I. and Angel S. (1977). *A Pattern Language.* New York: Oxford University Press
5. Apple Computer (1989). *MacApp II Programmer's Guide*; Cupertino, CA: Apple Computer, Inc.
6. Coplien J. and Schmidt D. (eds.) (1995). *Pattern Languages of Program Design.* Conference Proceedings. Reading, Massachusetts: Addison-Wesley
7. Coplien J.O. (1992). *Advanced C++ Programming Styles and Idioms.* Reading, Massachusetts: Addison-Wesley
8. Gamma E., Helm R., Johnson R., Vlissides J. (1995) *Design Patterns—Elements of Reusable Object-Oriented Software*; Reading, MA: Addison-Wesley.
9. Gamma E. (1991). *Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design*; doctoral thesis, University of Zürich, 1991; published by Springer Verlag, 1992.
10. Goldberg A., Rubin K. (1995). *Succeeding with Objects—Decision Frameworks for Project Management.* Reading, Massachusetts: Addison-Wesley
11. Helm R., Holland I.M. and Gangopadhyay D. (1990). Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA '90,* Ottawa, Canada
12. Johnson R.E. (1992). Documenting frameworks using patterns. In *Proceedings of OOPSLA '92,* Vancouver, Canada
13. Krasner G.E. and Pope S.T. (1988). A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming,* **1**(3)

14. Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K. (1996) *Object-Oriented Application Frameworks*. Manning Publications/Prentice Hall

15. Pree W. (1995). *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley/ACM Press

16. Pree W. (1996). *Framework Patterns*. New York City: SIGS Books

17. Pree W. (1997). *Komponentenbasierte Softwareentwicklung mit Frameworks*. Heidelberg: dpunkt.Verlag

18. Schmucker K. (1986). *Object-Oriented Programming for the Macintosh*. Hasbrouck Heights, NJ: Hayden

19. Taligent (1994). *Taligent's Guide to Designing Programs*. Reading, Massachusetts: Addison-Wesley

20. Weinand A., Gamma E., Marty R. (1988). *ET++ - An Object-Oriented Application Framework in C++*; OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11.

21. Wilson D.A., Rosenstein L.S. and Shafer D. (1990). *Programming with MacApp*. Reading, Massachusetts: Addison-Wesley