State-of-the-art Design Pattern Approaches— An Overview

Wolfgang Pree

C. Doppler Laboratory for Software Engineering
Johannes Kepler University Linz, A-4040 Linz, Austria
Voice: +43 70-2468-9432; Fax: +43 70-2468-9430
E-mail: pree@swe.uni-linz.ac.at

Abstract. Design patterns are considered as means to capture and communicate already proven and matured object-oriented design so that building reusable object-oriented software does not always have to start from scratch.

There exists no standardization of the term *design pattern* in the realm of object-oriented software development. This paper makes an effort to define this vague term by giving an overview of relevant state-of-the-art approaches.

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

1 Introduction

In general, patterns help to reduce the complexity in many real-life situations. For example, sometimes 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 by providing time-tested combinations that work. For example, car drivers apply a certain pattern to set a manual-transmission vehicle in motion. This balance of clutch and gas is applied no matter whether the vehicle is a Volkswagen Beetle or a Porsche 959.

Patterns appear in a vast variety of domains in daily life. In fashion, various patterns exist that describe good-looking combinations of various aspects like colors. More or less mandatory rules (for example, behavior patterns and traffic rules) might be derived from usable patterns.

Patterns are already well established in the realm of software development: 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. This paper deals only with these design patterns. From now on we use the terms pattern and design pattern synonymously with object-oriented design pattern and always mean the latter.

2 Categorization of design pattern approaches

As the design pattern approach in the realm of object-oriented software development has just emerged recently, there is no consensus on how to categorize design patterns. In order to categorize the design pattern approaches presented in this paper, we distinguish between the purpose of a particular design pattern approach and the applied notation.

Purpose. One of the principal goals of object-oriented software development is to improve the reusability of software components. Design patterns should help to improve reusability. Increased reusability of software is considered as crucial technical precondition to improve the overall software quality and reduce production and maintenance costs.

The concepts offered by object-oriented programming languages are sufficient to produce reusable single components and reusable frameworks. Frameworks consist of several single components and the bond between them, that is, their interaction. Frameworks represent application skeletons for a particular domain. So not only source code but also architecture design—which we consider as important characteristic of frameworks—is reused in applications built on top of frameworks.

The appropriate design of single components is a precondition for constructing frameworks. Some design pattern approaches focus on the design of single components or of a small group of components, ignoring the framework concept.

The design of reusable frameworks is more challenging. Thus we consider design pattern approaches as more advanced and more important if the framework aspect is stressed.

Notation. The following notations or combinations of them are applied in state-of-the-art design pattern approaches: informal textual notation; formal textual notation, that is, a formalism or a programming language; graphic notation.

By informal textual notation we mean a plain English description. For example, an informal description summarizes specific situations where a design pattern can be applied.

Code fragments written in a specific programming language only complement other notations. Bear in mind that design patterns have to abstract from programming language code, which should thus be reduced to a minimum in design pattern descriptions.

A graphic notation in the realm of object-oriented software development usually depicts class/object diagrams and complements the other notations.

Application to state-of-the-art approaches. Table 1 categorizes the design pattern approaches presented in this paper according to their purpose and notation. The Purpose column indicates which goals a particular design pattern approach pursues:

- “Components” rates how a particular design pattern approach focuses on the design of single components or of a small group of components, ignoring the concept of abstract classes and frameworks.

- "Frameworks I" rates how a particular design pattern approach focuses on achieving the goal of describing how to adapt a framework.
- "Frameworks II" rates how a particular design pattern approach focuses on achieving the goal of capturing the design of frameworks so that the design can be reused in the development of new frameworks.

We employ the ratings – (goal not pursued), ± (goal pursued to some degree) and + (goal pursued).

The Notation column lists notations applied in the various design pattern approaches. If one notation dominates another, it is listed before the less important one. For example, in the Design Pattern Catalog an informal textual notation and a graphic notation dominate by far the use of a programming language.

Table 1 Categorization of state-of-the-art design pattern approaches.

	Purpose		*Notation*
Object-Oriented Patterns (Peter Coad)	Components:	+	Informal textual notation
	Frameworks I:	–	Graphic notation
	Frameworks II:	±	
Coding Patterns	Components:	±	Programming language
	Frameworks I:	–	Informal textual notation
	Frameworks II:	–	
Framework Cookbooks	Components:	–	Informal textual notation
	Frameworks I:	+	Programming language
	Frameworks II:	–	
Design Pattern Catalog (Erich Gamma *et al.*)	Components:	±	Informal textual notation
	Frameworks I:	–	Graphic notation
	Frameworks II:	+	Programming language
Metapatterns (Wolfgang Pree)	Components:	±	Graphic notation
	Frameworks I:	+	Informal textual notation
	Frameworks II:	+	Programming language

3 Object-Oriented Patterns

According to Coad [1], design patterns are identified by observing the lowest-level building blocks, that is, classes and objects and the relationship established between them. We categorize Coad's patterns into

- basic inheritance and interaction patterns
- patterns for structuring object-oriented software systems
- patterns related to the MVC framework.

In order to give a glimpse of Coad's design pattern approach, we have chosen the Changeable Roles pattern as a representative pattern for structuring object-oriented software systems.

Changable Roles. Coad [1] motivates the significance of this pattern by means of an analogy. "A *player* object .... *wears different hats*, playing one or more *roles*."

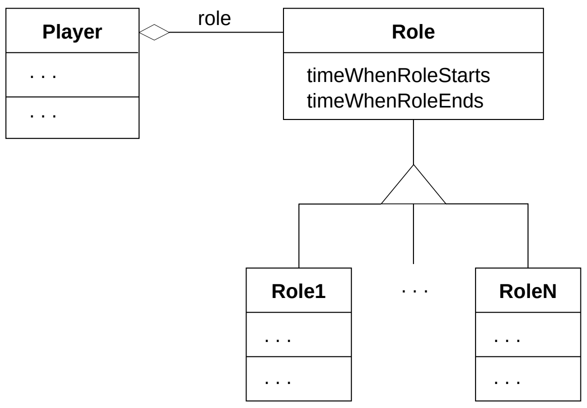

Figure 1 Design pattern facilitating changeable roles.

An instance of class Player refers to one Role object at a time. The Role object may be changed. Figure 1 shows the corresponding graphic representation in the Object Model Notation [9].

We consider another aspect important in the realm of this design pattern: class Role constitutes a typical example of an abstract class. Since the description of this pattern is too general, no hints can be given as to which behavior can be defined in class Role. The design pattern description only mentions that the two instance variables timeWhenRoleStarts and timeWhenRoleEnds are common attributes of various roles.

If this pattern is applied, additional common behavior should be defined in class Role as demonstrated in an enhanced version of this pattern in Section 6. Otherwise the implementation of Player will be cluttered with case statements where the role a Player object plays has to be checked. Which operations of the Role object to which the instance variable role refers can be invoked by a Player object depends on the result of the dynamic type check of instance variable role. This design would encumber extensions such as handling further Role objects in addition to the originally planned ones.

Summarizing Remarks. Most of Coad’s patterns suggest how to structure object-oriented software systems. Unfortunately, the importance of abstract classes and frameworks is not stressed—most pattern descriptions and examples do not mention these aspects, though some patterns are close to frameworks.

4 Coding Patterns

Coding patterns for C++ are discussed, for example, in [2, 10]. 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++.

Coding patterns depend on a particular programming language and/or class library. They primarily give basic hints on how to structure a software system from a syntactical point of view. These patterns are not suited to helping in the design of classes. The latter involves decisions about methods and instance variables grouped in a class, the meaning of the methods, and so on.

Obviously, coding patterns do not support a programmer in adapting or developing a framework either.

As we do not want to bother the reader with language-specific issues we refrain from presenting a representative example.

5 Framework adaptation patterns

This section illustrates *application framework cookbooks*. We use the short form *cookbook*. Cookbooks contain numerous *recipes*. They describe in an informal way how to adapt a framework in order to solve specific problems. Recipes usually do not explain the internal design and implementation details of a framework.

Recipes are rather informal documents. Nevertheless, most cookbook recipes are structured roughly into the sections purpose, procedure (including references to other recipes), and source code example(s).

A programmer has to find the recipe that is appropriate for a specific framework adaptation. A recipe is then used by simply adhering to the steps that describe how to accomplish a certain task. 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 [7] present a cookbook for using the MVC framework. We exemplify framework adaptation patterns by presenting fragments of a recipe for adapting the GUI framework ET++ [11]. We do not explain the context of this recipe, that is, the application domain and details such as classes mentioned in the recipe fragment.

> *Purpose*
>
> A Document object manages the data set of an application independently of how it is displayed or printed. It also provides methods to save the data in files and read it back (for details see the recipe "Saving and restoring data in documents").
>
> ET++ asks your application to create a new application-specific document when the user starts the application or chooses the New or Open menu commands.
>
> *Steps how to do it*
>
> (1) Declare the file type of your document as an instance of Symbol. This is usually done by an extern declaration in the header file where your Document class is defined:
>
> ```
> extern Symbol cYourDocType;
> ```
>
> The actual declaration is done in the implementation file of your Document class:
>
> ```
> Symbol cYourDocType("YourTypeName");
> ```

If you use a file format that is predefined in ET++, use the predefined file type. For example, a file made up of ASCII characters is of type cDocTypeAscii.

The file type has to be passed to the constructor of class Application. It becomes the principal file type of your application. If your application must deal with various file types, see the recipe “Handling different file types in an application”.

(2) ...

Summarizing Remarks. Cookbooks have to cover a wide range of typical framework adaptations, and they are hardly ever complete. The more adaptations are described in a cookbook, the harder it is to find the appropriate recipe. In many cases adaptations can be accomplished in several different ways. Cookbooks that encompass various different adaptations for solving a particular problem tend to become too complex: usually several adaptations have to be combined during application development using a framework. A programmer can easily become confused in searching for reasonable adaptation combinations.

In order to alleviate such problems, cookbooks have to be written by people who have an in-depth understanding of a framework. Ideally, of course, a cookbook is authored by the framework developers.

6 Design Pattern Catalog

Erich Gamma pioneered a catalog-like presentation of design patterns in his Ph.D. thesis [3]. Formal contracts [6] represent another attempt to describe framework design.

Gamma’s thesis built the basis of an advanced catalog-like presentation of more than 20 design patterns published in [4, 5]. We refer to the design patterns listed in these two publications as a *design pattern catalog* or simply *catalog*.

The design pattern catalog attaches importance to abstract classes and frameworks. Actually, most of the patterns constitute frameworks for a particular purpose. As a consequence, Gamma et al. [4] view these patterns as “reusable micro-architectures that contribute to an overall system architecture” that “... provide a common vocabulary for design.”

Each pattern in the catalog is described by means of informal text, one or more diagrams in the Object Model Notation, hints regarding the implementation, and code examples in C++ and/or Smalltalk. In essence, a pattern description starts with a sample usage of a particular pattern in a certain context. Usually a graphic diagram and an informal textual description provide a brief overview of the pattern and show the advantages offered by the pattern. A more abstract informal description of the pattern participants and their collaborations follows, accompanied by a corresponding class/object diagram. Relevant clues outlining typical situations when to use the pattern, including caveats, follow the abstract description. Code examples and examples of object-oriented systems where the pattern was successfully applied conclude the pattern description. References to other pattern descriptions point out related patterns and their differences.

In order to give a glimpse of which entries can be found in the design pattern catalog we pick out the State pattern and summarize this entry.

State Pattern. The State pattern in the catalog is similar to Coad's Changeable Roles pattern described in Section 3. Recall the remark regarding the Changeable Roles pattern: class Role should be an abstract class. Coad's pattern is considered to be too general to give hints on which behavior should be defined in the abstract class Role. The State pattern eliminates this deficiency.

Figure 2 shows the enhanced Changeable Roles pattern. In the design pattern catalog, class Player is called Context; class Role corresponds to class State.

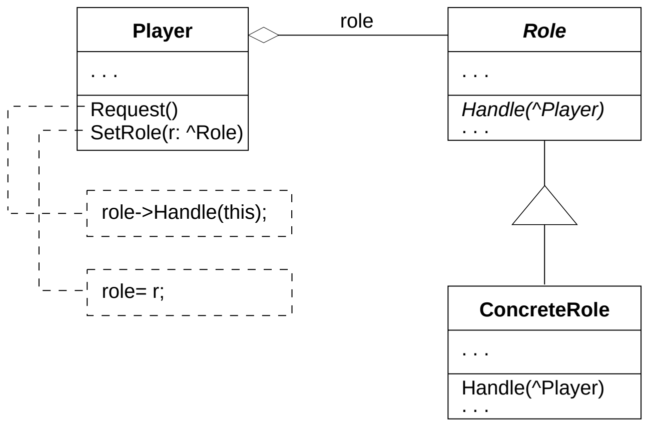

Figure 2 State pattern as an enhanced Changeable Roles pattern.

The design pattern catalog outlines a sample usage of this pattern in the realm of networking: a class TCPConnection corresponds to Player, an abstract class TCPState to Role. TCPClosed and TCPEstablished are two subclasses of TCPState. Depending on the actual object referred to by the instance variable role, an object of class TCPConnection handles requests differently, like opening a connection.

Analogous Patterns. Numerous other patterns in the catalog are based on abstract coupling. Typically, the semantics, that is, the method names and the entities represented by the classes, differ in these patterns. For example, in the State pattern the classes Player and Role cooperate via the Handle method provided by class Role. The part that is kept flexible in this pattern is the way a request is handled by a specific Role object.

Summarizing Remarks. Most of the patterns focus on frameworks. Gamma et al. [4] point out the difference between frameworks and design patterns: ".... frameworks are implemented in a programming language. .... In this sense frameworks are more concrete than design patterns. .... Mature frameworks usually reuse several design patterns."

Since experienced object-oriented designers collected these patterns, the catalog represents a valuable resource of micro-architectures. Knowing these design examples can help in the development of new frameworks.

Nevertheless, numerous patterns in the catalog are similar. In the following section we present a more advanced abstraction that allows to categorize the catalog patterns and thus forms the basis to build adequate tools.

7 Metapatterns

We introduce the term *metapatterns* for a set of design patterns that describes how to construct frameworks independent of a specific domain. This more advanced abstraction is helpful, for example, in order to actively support the design pattern idea in the realm of tools. Actually, constructing frameworks by combining the basic object-oriented concepts proves quite straightforward.

Hot Spots and Frozen Spots. In general, an application framework standardizes applications for a specific domain. Usually, various aspects of an application framework cannot be anticipated. These parts of an application framework have to be generic so that they can easily be adapted to specific needs. Figure 3 shows schematically this property of an application framework with the generic, flexible parts in gray.

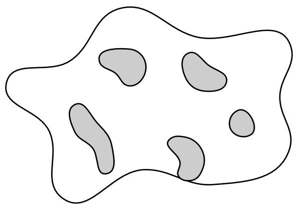

Figure 3 An application framework with flexible hot spots (gray).

The difficulty of good object-oriented design is to identify the *hot spots* of an application framework, that is, those aspects of an application domain that have to be kept flexible. We consider a framework to have the quality attribute “well designed” if it provides adequate hot spots for adaptations. The other parts of a framework comprise the standardized behavior corresponding to *frozen spots* (depicted as white cloud in Figure 3).

Primarily, *domain-specific knowledge is required* to find these hot spots. Only domain analysis can help to acquire this knowledge. However, framework examples and metapatterns are quite useless during this domain analysis.

Once hot spots and the corresponding desired degree of flexibility are identified metapatterns describe how to design the particular parts of the overall software system that matches these hot spots.

Overall seven metapatterns have been identified. [8] describes these metapatterns in detail together with their characteristics (such as the degree of flexibility and the structure of methods) Hints are given when to choose a particular pattern. Below we pick out the simplest one in order to illustrate the idea.

Unification pattern. This metapattern represents the one with the lowest degree of flexibility. In order to change the hot spot behavior the system has to be restarted. Run-time changes are not possible.

In the Unification pattern the template method (corresponding to a frozen spot) and the hook method (corresponding to a hot spot) are unified in one class.

Let us consider the example shown in Figure 4 (a): a class B offers three methods M1(), M2() and M3(). M1() constitutes the template method based on the hook methods M2() and M3(). For method M2() only the method interface, that is, its name and parameters, can be defined, not an implementation. Such abstract methods are written in *italic style* in the graphic representation. Method M3() is assumed to provide a meaningful default implementation.

Class B could, for example, correspond to a class RentalItem, which could be part of an application framework for reservation systems (see Figure 4 (b)). Class B's template method M1() corresponds to PrintInvoice() of class RentalItem. For method CalcRate() only the method interface can be defined, not an implementation. Method GetName() is assumed to provide a meaningful default implementation.

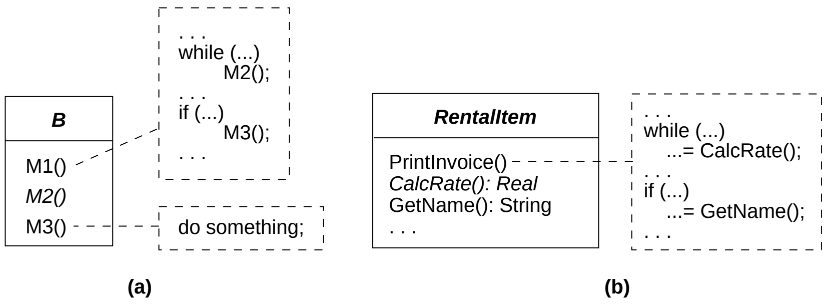

Figure 4 Template and hook methods unified in one class.

B must be adapted in a subclass where at least the abstract method M2() of B is overridden. The default implementations of M1() and M3() hopefully meet the requirements of the specific application under development. RentalItem is adapted in an analogous way.

Metapattern Annotations. The seven metapatterns presented in [8] repeatedly occur in frameworks. Of course, each framework uses specific names for the templates and hooks. But the core characteristics of the metapatterns are independent of their particular application. Figure 5 depicts such an annotation schematically by means of arrows.

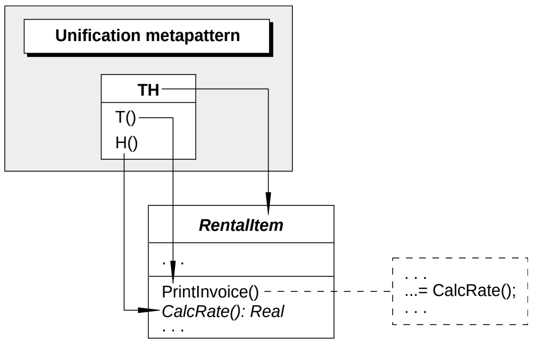

Figure 5 A hot spot based on the Unification metapattern.

In an appropriate hypertext system each of the seven metapatterns might be linked to numerous components of a framework; that is, the components of a metapattern are linked to domain-specific templates and hooks. Thus metapatterns are a means of capturing and communicating the design incorporated in a framework via metapattern browsers. Metapattern browsers visualize the hot spots of a framework.

Implications. Metapattern browsers for mature frameworks can be viewed as advanced design pattern catalogs. Some aspects of a specific framework might be domain-independent to a large degree, so that this design can be applied in the development of new frameworks. In these cases metapattern browsers serve the same purpose as design pattern catalogs.

Compared to design pattern catalogs, metapattern browsers can additionally annotate any domain-specific framework and document its design. The fact that metapattern browsers allow efficient design documentation of frameworks can help in adapting the hot spots of a framework to specific needs.

Future research based on a prototype implementation of a metapattern browser will reveal the suitability of a design documentation based on metapatterns for the adaptation and development of frameworks.

8 Conclusion

This paper outlined characteristics of state-of-the-art design pattern approaches as a means of capturing and communicating the design of object-oriented systems, especially frameworks.

Only frameworks allow to fully exploit the potential of object-oriented software development. Design patterns recently emerged as a glimmer of hope on the horizon for supporting the development and reuse of frameworks. OOAD methodologies assist in the development of a well-structured object-oriented system. Frameworks have to evolve from this initial framework design. Design patterns can support this architecture evolution. In that sense, OOAD methodologies are complemented by design pattern approaches.

References

1. Coad P.: Object-Oriented Patterns; in Communications of the ACM, Vol. 33, No. 9, Sept. 1992.
2. Coplien J.O.: Advanced C++ Programming Styles and Idioms; Reading, Massachusetts: Addison-Wesley, 1992.
3. Gamma E.: Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design; doctoral thesis, University of Zürich, 1991; published by Springer Verlag, 1992.
4. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Abstraction and Reuse of Object-Oriented Design; in ECOOP'93 Conference Proceedings, Springer Verlag, 1993.
5. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns—Elements of Reusable Object-Oriented Software; Addison-Wesley, 1995.
6. Helm R., Holland I.M. and Gangopadhyay D.: Contracts: specifying behavioral compositions in object-oriented systems. In Proceedings of OOPSLA '90, Ottawa, Canada, 1990.

7. Krasner G.E. and Pope S.T.: A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, **1**(3), 1988.

8. Pree W.: Design Patterns for Object-Oriented Software Development; Addison-Wesley/ACM Press, 1995.

9. Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modeling and Design; Prentice Hall, Englewood Cliffs, New Jersey, 1991.

10. Taligent: Taligent's Guide to Designing Programs—Well-Mannered Object-Oriented Design in C++; Reading, Massachusetts: Addison-Wesley, 1994.

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