Design Patterns: Abstraction and Reuse of Object-Oriented Design

Erich Gamma[1*], Richard Helm[2], Ralph Johnson[3], John Vlissides[2]

[1] Taligent, Inc.
10725 N. De Anza Blvd., Cupertino, CA 95014-2000 USA

[2] I.B.M. Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598 USA

[3] Department of Computer Science
University of Illinois at Urbana-Champaign
1034 W. Springfield Ave., Urbana, IL 61801 USA

Abstract. We propose **design patterns** as a new mechanism for expressing object-oriented design experience. Design patterns identify, name, and abstract common themes in object-oriented design. They capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities. Design patterns play many roles in the object-oriented development process: they provide a common vocabulary for design, they reduce system complexity by naming and defining abstractions, they constitute a base of experience for building reusable software, and they act as building blocks from which more complex designs can be built. Design patterns can be considered reusable micro-architectures that contribute to an overall system architecture. We describe how to express and organize design patterns and introduce a catalog of design patterns. We also describe our experience in applying design patterns to the design of object-oriented systems.

1 Introduction

Design methods are supposed to promote good design, to teach new designers how to design well, and to standardize the way designs are developed. Typically, a design method comprises a set of syntactic notations (usually graphical) and a set of rules that govern how and when to use each notation. It will also describe problems that occur in a design, how to fix them, and how to evaluate a design. Studies of expert programmers for conventional languages, however, have shown that knowledge is not organized simply around syntax, but in larger conceptual structures such as algorithms, data structures and idioms [1, 7, 9, 27], and plans that indicate steps necessary to fulfill a particular goal [26]. It is likely that designers do not think about the notation they are using for recording the design. Rather, they look for patterns to match against plans, algorithms, data structures, and idioms they have learned in the past. Good designers, it appears, rely on large amounts of design experience, and this experience is just as important as the notations for recording designs and the rules for using those notations.

Our experience with the design of object-oriented systems and frameworks [15, 17, 22, 30, 31] bears out this observation. We have found that there exist idiomatic class and object structures that help make designs more flexible, reusable, and elegant. For example, the Model-View-Controller (MVC) paradigm from Smalltalk [19] is a design structure that separates representation from presentation. MVC promotes flexibility in the choice of views, independent of the model. Abstract factories [10] hide concrete subclasses from the applications that use them so that class names are not hard-wired into an application.

---

[*] Work performed while at UBILAB, Union Bank of Switzerland, Zurich, Switzerland.

To appear in *ECOOP '93 Conference Proceedings, Springer-Verlag Lecture Notes in Computer Science.*

Well-defined design structures like these have a positive impact on software development. A software architect who is familiar with a good set of design structures can apply them immediately to design problems without having to rediscover them. Design structures also facilitate the reuse of successful architectures—expressing proven techniques as design structures makes them more readily accessible to developers of new systems. Design structures can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent.

To this end we propose **design patterns**, a new mechanism for expressing design structures. Design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations, and the distribution of responsibilities. Design patterns have many uses in the object-oriented development process:

- Design patterns provide a common vocabulary for designers to communicate, document, and explore design alternatives. They reduce system complexity by naming and defining abstractions that are above classes and instances. A good set of design patterns effectively raises the level at which one programs.
- Design patterns constitute a reusable base of experience for building reusable software. They distill and provide a means to reuse the design knowledge gained by experienced practitioners. Design patterns act as building blocks for constructing more complex designs; they can be considered **micro-architectures** that contribute to overall system architecture.
- Design patterns help reduce the learning time for a class library. Once a library consumer has learned the design patterns in one library, he can reuse this experience when learning a new class library. Design patterns help a novice perform more like an expert.
- Design patterns provide a target for the reorganization or refactoring of class hierarchies [23]. Moreover, by using design patterns early in the lifecycle, one can avert refactoring at later stages of design.

The major contributions of this paper are: a definition of design patterns, a means to describe them, a system for their classification, and most importantly, a catalog containing patterns we have discovered while building our own class libraries and patterns we have collected from the literature. This work has its roots in Gamma's thesis [11], which abstracted design patterns from the ET++ framework. Since then the work has been refined and extended based on our collective experience. Our thinking has also been influenced and inspired by discussions within the Architecture Handbook Workshops at recent OOPSLA conferences [3, 4].

This paper has two parts. The first introduces design patterns and explains techniques to describe them. Next we present a classification system that characterizes common aspects of patterns. This classification will serve to structure the catalog of patterns presented in the second part of this paper. We discuss how design patterns impact object-oriented programming and design. We also review related work.

The second part of this paper (the Appendix) describes our current catalog of design patterns. As we cannot include the complete catalog in this paper (it currently runs over 90 pages [12]), we give instead a brief summary and include a few abridged patterns. Each pattern in this catalog is representative of what we judge to be good object-oriented design. We have tried to reduce the subjectivity of this judgment by including only design patterns that have seen practical application. Every design pattern we have included works—most have been used at least twice and have either been discovered independently or have been used in a variety of application domains.

2 Design Patterns

A design pattern consists of three essential parts:

1. An abstract description of a class or object collaboration and its structure. The description is abstract because it concerns abstract design, not a particular design.
2. The issue in system design addressed by the abstract structure. This determines the circumstances in which the design pattern is applicable.
3. The consequences of applying the abstract structure to a system's architecture. These determine if the pattern should be applied in view of other design constraints.

Design patterns are defined in terms of object-oriented concepts. They are sufficiently abstract to avoid specifying implementation details, thereby ensuring wide applicability, but a pattern may provide hints about potential implementation issues.

We can think of a design pattern as a micro-architecture. It is an architecture in that it serves as a blueprint that may have several realizations. It is "micro" in that it defines something less than a complete application or library. To be useful, a design pattern should be applicable to more than a few problem domains; thus design patterns tend to be relatively small in size and scope. A design pattern can also be considered a transformation of system structure. It defines the context for the transformation, the change to be made, and the consequences of this transformation.

To help readers understand patterns, each entry in the catalog also includes detailed descriptions and examples. We use a template (Figure 1) to structure our descriptions and to ensure uniformity between entries in the catalog. This template also explains the motivation behind its structure. The Appendix contains three design patterns that use the template. We urge readers to study the patterns in the Appendix as they are referenced in the following text.

3 Categorizing Design Patterns

Design patterns vary in their granularity and level of abstraction. They are numerous and have common properties. Because there are many design patterns, we need a way to organize them. This section introduces a classification system for design patterns. This classification makes it easy to refer to families of related patterns, to learn the patterns in the catalog, and to find new patterns.

		Characterization		
		Creational	**Structural**	**Behavioral**
Jurisdiction	**Class**	Factory Method	Adapter (class) Bridge (class)	Template Method
	Object	Abstract Factory Prototype Solitaire	Adapter (object) Bridge (object) Flyweight Glue Proxy	Chain of Responsibility Command Iterator (object) Mediator Memento Observer State Strategy
	Compound	Builder	Composite Wrapper	Interpreter Iterator (compound) Walker

Table 1. Design Pattern Space

DESIGN PATTERN NAME Jurisdiction Characterization

What is the pattern's name and classification? The name should convey the pattern's essence succinctly. A good name is vital, as it will become part of the design vocabulary.

Intent

What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Motivation

A scenario in which the pattern is applicable, the particular design problem or issue the pattern addresses, and the class and object structures that address this issue. This information will help the reader understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can one recognize these situations?

Participants

Describe the classes and/or objects participating in the design pattern and their responsibilities using CRC conventions [5].

Collaborations

Describe how the participants collaborate to carry out their responsibilities.

Diagram

A graphical representation of the pattern using a notation based on the Object Modeling Technique (OMT) [25], to which we have added method pseudo-code.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What does the design pattern objectify? What aspect of system structure does it allow to be varied independently?

Implementation

What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there language-specific issues?

Examples

This section presents examples from real systems. We try to include at least two examples from different domains.

See Also

What design patterns have closely related intent? What are the important differences? With which other patterns should this one be used?

Fig. 1. Basic Design Pattern Template

We can think of the set of all design patterns in terms of two orthogonal criteria, **jurisdiction** and **characterization**. Table 1 organizes our current set of patterns according to these criteria.

Jurisdiction is the domain over which a pattern applies. Patterns having **class** jurisdiction deal with relationships between base classes and their subclasses; class jurisdiction covers static semantics. The **object** jurisdiction concerns relationships between peer objects. Patterns having **compound** jurisdiction deal with recursive object structures. Some patterns capture concepts that span jurisdictions. For example, iteration applies both to collections of objects (i.e., object jurisdiction) and to recursive object structures (compound jurisdiction). Thus there are both object and compound versions of the Iterator pattern.

Characterization reflects what a pattern does. Patterns can be characterized as either **creational**, **structural**, or **behavioral**. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The following sections describe pattern jurisdictions in greater detail for each characterization using examples from our catalog.

3.1 Class Jurisdiction

Class Creational. Creational patterns abstract how objects are instantiated by hiding the specifics of the creation process. They are useful because it is often undesirable to specify a class name explicitly when instantiating an object. Doing so limits flexibility; it forces the programmer to commit to a particular class instead of a particular protocol. If one avoids hard-coding the class, then it becomes possible to defer class selection to run-time.

Creational class patterns in particular defer some part of object creation to subclasses. An example is the Factory Method, an abstract method that is called by a base class but defined in subclasses. The subclass methods create instances whose type depends on the subclass in which each method is implemented. In this way the base class does not hard-code the class name of the created object. Factory Methods are commonly used to instantiate members in base classes with objects created by subclasses.

For example, an abstract Application class needs to create application-specific documents that conform to the Document type. Application instantiates these Document objects by calling the factory method DoMakeDocument. This method is overridden in classes derived from Application. The subclass DrawApplication, say, overrides DoMakeDocument to return a DrawDocument object.

Class Structural. Structural class patterns use inheritance to compose protocols or code. As a simple example, consider using multiple inheritance to mix two or more classes into one. The result is an amalgam class that unites the semantics of the base classes. This trivial pattern is quite useful in making independently-developed class libraries work together [15].

Another example is the class-jurisdictional form of the Adapter pattern. In general, an Adapter makes one interface (the adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces. A class Adapter accomplishes this by inheriting privately from an adaptee class. The Adapter then expresses its interface in terms of the adaptee's.

Class Behavioral. Behavioral class patterns capture how classes cooperate with their subclasses to fulfill their semantics. Template Method is a simple and well-known behavioral class pattern [32]. Template methods define algorithms step by step. Each step can invoke an abstract method (which the subclass must define) or a base method. The purpose of a template method is to provide an abstract definition of an algorithm. The subclass must implement specific behavior to provide the services required by the algorithm.

3.2 Object Jurisdiction

Object patterns all apply various forms of non-recursive object composition. Object composition represents the most powerful form of reusability—a collection of objects are most easily reused through variations on how they are composed rather than how they are subclassed.

Object Creational. Creational object patterns abstract how sets of objects are created. The Abstract Factory pattern (page 14) is a creational object pattern. It describes how to create "product" objects through an generic interface. Subclasses may manufacture specialized versions or compositions of objects as permitted by this interface. In turn, clients can use abstract factories to avoid making assumptions about what classes to instantiate. Factories can be composed to create larger factories whose structure can be modified at run-time to change the semantics of object creation. The factory may manufacture a custom composition of instances, a shared or one-of-a-kind instance, or anything else that can be computed at run-time, so long as it conforms to the abstract creation protocol.

For example, consider a user interface toolkit that provides two types of scroll bars, one for Motif and another for Open Look. An application programmer may not want to hard-code one or the other into the application—the choice of scroll bar will be determined by, say, an environment variable. The code that creates the scroll bar can be encapsulated in the class **Kit**, an abstract factory that abstracts the specific type of scroll bar to instantiate. Kit defines a protocol for creating scroll bars and other user interface elements. Subclasses of Kit redefine operations in the protocol to return specialized types of scroll bars. A MotifKit's scroll bar operation would instantiate and return a Motif scroll bar, while the corresponding OpenLookKit operation would return an Open Look scroll bar.

Object Structural. Structural object patterns describe ways to assemble objects to realize new functionality. The added flexibility inherent in object composition stems from the ability to change the composition at run-time, which is impossible with static class composition.

Proxy is an example of a structural object pattern. A proxy acts as a convenient surrogate or placeholder for another object. A proxy can be used as a local representative for an object in a different address space (remote proxy), to represent a large object that should be loaded on demand (virtual proxy), or to protect access to the original object (protected proxy). Proxies provide a level of indirection to particular properties of objects. Thus they can restrict, enhance, or alter an object's properties.

The Flyweight pattern is concerned with object sharing. Objects are shared for at least two reasons: efficiency and consistency. Applications that use large quantities of objects must pay careful attention to the cost of each object. Substantial savings can accrue by sharing objects instead of replicating them. However, objects can only be shared if they do not define context-dependent state. Flyweights have no context-dependent state. Any additional information they need to perform their task is passed to them when needed. With no context-dependent state, flyweights may be shared freely. Moreover, it may be necessary to ensure that all copies of an object stay consistent when one of the copies changes. Sharing provides an automatic way to maintain this consistency.

Object Behavioral. Behavioral object patterns describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. For example, patterns such as Mediator and Chain of Responsibility abstract control flow. They call for objects that exist solely to redirect the flow of messages. The redirection may simply notify another object, or it may involve complex computation and buffering. The Observer pattern abstracts the synchronization of state or behavior. Entities that are co-dependent to the extent that their state must remain synchronized may exploit Observer. The classic example is the model-view pattern, in which multiple views of the model are notified whenever the model's state changes.

The Strategy pattern (page 17) objectifies an algorithm. For example, a text composition object may need to support different line breaking algorithms. It is infeasible to hard-wire all such algorithms into the text composition class and subclasses. An alternative is to objectify different algorithms and

However, object models that support dynamic inheritance, most notably Self [29], are as flexible as object composition in theory.

provide them as **Compositor** subclasses. The interface for Compositors is defined by the abstract Compositor class, and its derived classes provide different layout strategies, such as simple line breaks or full page justification. Instances of the Compositor subclasses can be coupled with the text composition at run-time to provide the appropriate text layout. Whenever a text composition has to find line breaks, it forwards this responsibility to its current Compositor object.

3.3 Compound Jurisdiction

In contrast to patterns having object jurisdiction, which concern peer objects, patterns with compound jurisdiction affect recursive object structures.

Compound Creational. Creational compound patterns are concerned with the creation of recursive object structures. An example is the Builder pattern. A Builder base class defines a generic interface for incrementally constructing recursive object structures. The Builder hides details of how objects in the structure are created, represented, and composed so that changing or adding a new representation only requires defining a new Builder class. Clients will be unaffected by changes to Builder.

Consider a parser for the RTF (Rich Text Format) document exchange format that should be able to perform multiple format conversions. The parser might convert RTF documents into (1) plain ASCII text and (2) a text object that can be edited in a text viewer object. The problem is how to make the parser independent of these different conversions.

The solution is to create an RTFReader class that takes a Builder object as an argument. The RTFReader knows how to parse the RTF format and notifies the Builder whenever it recognizes text or an RTF control word. The builder is responsible for creating the corresponding data structure. It separates the parsing algorithm from the creation of the structure that results from the parsing process. The parsing algorithm can then be reused to create any number of different data representations. For example, an ASCII builder ignores all notifications except plain text, while a Text builder uses the notifications to create a more complex text structure.

Compound Structural. Structural compound patterns capture techniques for structuring recursive object structures. A simple example is the Composite pattern. A Composite is a recursive composition of one or more other Composites. A Composite treats multiple, recursively composed objects as a single object.

The Wrapper pattern (page 20) describes how to flexibly attach additional properties and services to an object. Wrappers can be nested recursively and can therefore be used to compose more complex object structures. For example, a Wrapper containing a single user interface component can add decorations such as borders, shadows, scroll bars, or services like scrolling and zooming. To do this, the Wrapper must conform to the interface of its wrapped component and forward messages to it. The Wrapper can perform additional actions (such as drawing a border around the component) either before or after forwarding a message.

Compound Behavioral. Finally, behavioral compound patterns deal with behavior in recursive object structures. Iteration over a recursive structure is a common activity captured by the Iterator pattern. Rather than encoding and distributing the traversal strategy in each class in the structure, it can be extracted and implemented in an Iterator class. Iterators objectify traversal algorithms over recursive structures. Different iterators can implement pre-order, in-order, or post-order traversals. All that is required is that nodes in the structure provide services to enumerate their sub-structures. This avoids hard-wiring traversal algorithms throughout the classes of objects in a composite structure. Iterators may be replaced at run-time to provide alternative traversals.

4 Experience with Design Patterns

We have applied design patterns to the design and construction of a several systems. We briefly describe two of these systems and our experience.

4.1 ET++SwapsManager

The ET++SwapsManager [10] is a highly interactive tool that lets traders value, price, and perform what-if analyses for a financial instrument called a **swap**. During this project the developers had to first learn the ET++ class library, then implement the tool, and finally design a framework for creating "calculation engines" for different financial instruments. While teaching ET++ we emphasized not only learning the class library but also describing the applied design patterns. We noticed that design patterns reduced the effort required to learn ET++. Patterns also proved helpful during development in design and code reviews. Patterns provided a common vocabulary to discuss a design. Whenever we encountered problems in the design, patterns helped us explore design alternatives and find solutions.

4.2 QOCA: A Constraint Solving Toolkit

QOCA (Quadratic Optimization Constraint Architecture) [14, 15] is a new object-oriented constraint-solving toolkit developed at IBM Research. QOCA leverages recent results in symbolic computation and geometry to support efficient incremental and interactive constraint manipulation. QOCA's architecture is designed to be flexible. It permits experimentation with different classes of constraints and domains (e.g., reals, booleans, etc.), different constraint-solving algorithms for these domains, and different representations (doubles, infinite precision) for objects in these domains. QOCA's object-oriented design allows parts of the system to be varied independently of others. This flexibility was achieved, for example, by using Strategy patterns to factor out constraint solving algorithms and Bridges to factor out domains and representations of variables. In addition, the Observable pattern is used to propagate notifications when variables change their values.

4.3 Summary of Observations

The following points summarize the major observations we have made while applying design patterns:

- Design patterns motivate developers to go beyond concrete objects; that is, they objectify concepts that are not immediately apparent as objects in the problem domain.
- Choosing intuitive class names is important but also difficult. We have found that design patterns can help name classes. In the ET++SwapsManager's calculation engine framework we encoded the name of the design pattern in the class name (for example CalculationStrategy or TableAdaptor). This convention results in longer class names, but it gives clients of these classes a hint about their purpose.
- We often apply design patterns *after* the first implementation of an architecture to improve its design. For example, it is easier to apply the Strategy pattern after the initial implementation to create objects for more abstract notions like a calculation engine or constraint solver. Patterns were also used as targets for class refactorings. We often find ourselves saying, "Make this part of a class into a Strategy," or, "Let's split the implementation portion of this class into a Bridge."
- Presenting design patterns together with examples of their application turned out to be an effective way to teach object-oriented design by example.
- An important issue with any reuse technology is how a reusable component can be adapted to create a problem-specific component. Design patterns are particularly suited to reuse because they are abstract. Though a concrete class structure may not be reusable, the design pattern underlying it often is.

– Design patterns also reduce the effort required to learn a class library. Each class library has a certain design "culture" characterized by the set of patterns used implicitly by its developers. A specific design pattern is typically reused in different places in the library. A client should therefore learn these patterns as a first step in learning the library. Once they are familiar with the patterns, they can reuse this understanding. Moreover, because some patterns appear in other class libraries, it is possible to reuse the knowledge about patterns when learning other libraries as well.

5 Related Work

Design patterns are an approach to software reuse. Krueger [20] introduces the following taxonomy to characterize different reuse approaches: software component reuse, software schemas, application generators, transformation systems, and software architectures. Design patterns are related to both software schemas and reusable software architectures. Software schemas emphasize reusing abstract algorithms and data structures. These abstractions are represented formally so they can be instantiated automatically. The Paris system [18] is representative of schema technology. Design patterns are higher-level than schemas; they focus on design structures at the level of collaborating classes and not at the algorithmic level. In addition, design patterns are not formal descriptions and cannot be instantiated directly. We therefore prefer to view design patterns as reusable software architectures. However, the examples Krueger lists in this category (blackboard architectures for expert systems, adaptable database subsystems) are all coarse-grained architectures. Design patterns are finer-grained and therefore can be characterized as reusable micro-architectures.

Most research into patterns in the software engineering community has been geared towards building knowledge-based assistants for automating the application of patterns for synthesis (that is, to write programs) and analysis (in debugging, for example) [13, 24]. The major difference between our work and that of the knowledge-based assistant community is that design patterns encode higher-level expertise. Their work has tended to focus on patterns like enumeration and selection, which can be expressed directly as reusable components in most existing object-oriented languages. We believe that characterizing and cataloging higher-level patterns that designers already use informally has an immediate benefit in teaching and communicating designs.

A common approach for reusing object-oriented software architectures are object-oriented frameworks [32]. A framework is a codified architecture for a problem domain that can be adapted to solve specific problems. A framework makes it possible to reuse an architecture together with a partial concrete implementation. In contrast to frameworks, design patterns allow only the reuse of abstract micro-architectures without a concrete implementation. However, design patterns can help define and develop frameworks. Mature frameworks usually reuse several design patterns. An important distinction between frameworks and design patterns is that frameworks are implemented in a programming language. Our patterns are ways of *using* a programming language. In this sense frameworks are more concrete than design patterns.

Design patterns are also related to the idioms introduced by Coplien [7]. These idioms are concrete design solutions in the context of C++. Coplien "focuses on idioms that make C++ programs more expressive." In contrast, design patterns are more abstract and higher-level than idioms. Patterns try to abstract design rather than programming techniques. Moreover, design patterns are usually independent of the implementation language.

There has been interest recently within the object-oriented community [8] in pattern languages for the architecture of buildings and communities as advocated by Christopher Alexander in *The Timeless Way of Building* [2]. Alexander's patterns consist of three parts:

– A context that describes when a pattern is applicable.
– The problem (or "system of conflicting forces") that the pattern resolves in that context.
– A configuration that describes physical relationships that solve the problem.

Both design patterns and Alexander's patterns share the notion of context/problem/configuration, but our patterns currently do not form a complete system of patterns and so do not strictly define a pattern language. This may be because object-oriented design is still a young technology—we may not have had enough experience in what constitutes good design to extract design patterns that cover all phases of the design process. Or this may be simply because the problems encountered in software design are different from those found in architecture and are not amenable to solution by pattern languages.

Recently, Johnson has advocated pattern languages to describe how to use use object-oriented frameworks [16]. Johnson uses a pattern language to explain how to extend and customize the Hotdraw drawing editor framework. However, these patterns are not design patterns; they are more descriptions of how to reuse existing components and frameworks instead of rules for generating new designs.

Coad's recent paper on object-oriented patterns [6] is also motivated by Alexander's work but is more closely related to our work. The paper has seven patterns: "Broadcast" is the same as Observer, but the other patterns are different from ours. In general, Coad's patterns seem to be more closely related to analysis than design. Design patterns like Wrapper and Flyweight are unlikely to be generated naturally during analysis unless the analyst knows these patterns well and thinks in terms of them. Coad's patterns could naturally arise from a simple attempt to model a problem. In fact, it is hard to see how any large model could avoid using patterns like "State Across a Collection" (which explains how to use aggregation) or "Behavior Across a Collection" (which describes how to distribute responsibility among objects in an aggregate). The patterns in our catalog are typical of a mature object-oriented design, one that has departed from the original analysis model in an attempt to make a system of reusable objects. In practice, both types of patterns are probably useful.

6 Conclusion

Design patterns have revolutionized the way we think about, design, and teach object-oriented systems. We have found them applicable in many stages of the design process—initial design, reuse, refactoring. They have given us a new level of abstraction for system design.

New levels of abstraction often afford opportunities for increased automation. We are investigating how interactive tools can take advantage of design patterns. One of these tools lets a user explore the space of objects in a running program and watch their interaction. Through observation the user may discover existing or entirely new patterns; the tool lets the user record and catalog his observations. The user may thus gain a better understanding of the application, the libraries on which it is based, and design in general.

Design patterns may have an even more profound impact on how object-oriented systems are designed than we have discussed. Common to most patterns is that they permit certain aspects of a system to be varied independently. This leads to thinking about design in terms of "What aspect of a design should be variable?" Answers to this question lead to certain applicable design patterns, and their application leads subsequently to modification of a design. We refer to this design activity as **variation-oriented design** and discuss it more fully in the catalog of patterns [12].

But some caveats are in order. Design patterns should not be applied indiscriminately. They typically achieve flexibility and variability by introducing additional levels of indirection and can therefore complicate a design. A design pattern should only be applied when the flexibility it affords is actually needed. The consequences described in a pattern help determine this. Moreover, one is often tempted to brand any new programming trick a new design pattern. A true design pattern will be non-trivial and will have had more than one application.

We hope that the design patterns described in this paper and in the companion catalog will provide the object-oriented community both a common design terminology and a repertoire of reusable designs. Moreover, we hope the catalog will motivate others to describe their systems in terms of design patterns and develop their own design patterns for others to reuse.

7 Acknowledgements

The authors wish to thank Doug Lea and Kent Beck for detailed comments and discussions about this work, and Bruce Anderson and the participants of the Architecture Handbook workshops at OOPSLA '91 and '92.

References

1. B. Adelson and Soloway E. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11(11):1351–1360, 1985.
2. Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.
3. Association for Computing Machinery. *Addendum to the Proceedings, Object-Oriented Programming Systems, Languages, and Applications Conference*, Phoenix, AZ, October 1991.
4. Association for Computing Machinery. *Addendum to the Proceedings, Object-Oriented Programming Systems, Languages, and Applications Conference*, Vancouver, British Columbia, October 1992.
5. Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 1–6, New Orleans, LA, October 1989.
6. Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, September 1992.
7. James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, Reading, Massechusetts, 1992.
8. Ward Cunningham and Kent Beck. Constructing abstractions for object-oriented applications. Technical Report CR-87-25, Computer Research Laboratory, Tektronix, Inc., 1987.
9. Bill Curtis. Cognitive issues in reusing software artifacts. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume II*, pages 269–287. Addison-Wesley, 1989.
10. Thomas Eggenschwiler and Erich Gamma. The ET++SwapsManager: Using object technology in the financial engineering domain. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 166–178, Vancouver, British Columbia, October 1992.
11. Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge.* Springer-Verlag, Berlin, 1992.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. A catalog of object-oriented design patterns. Technical Report in preparation, IBM Research Division, 1992.
13. Mehdi T. Harandi and Frank H. Young. Software design using reusable algorithm abstraction. In *In Proc. 2nd IEEE/BCS Conf. on Software Engineering*, pages 94–97, 1985.
14. Richard Helm, Tien Huynh, Catherine Lassez, and Kim Marriott. A linear constraint technology for user interfaces. In *Graphics Interface*, pages 301–309, Vancouver, British Columbia, 1992.
15. Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, pages 1–22, Champéry, Switzerland, October 1992. Also available as IBM Research Division Technical Report RC 18524 (79392).
16. Ralph Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 63–76, Vancouver, BC, October 1992.
17. Ralph E. Johnson, Carl McConnell, and J. Michael Lake. The RTL system: A framework for code optimization. In Robert Giegerich and Susan L. Graham, editors, *Code Generation—Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, 1992. Springer-Verlag.
18. S. Katz, C.A. Richter, and K.-S. The. Paris: A system for reusing partially interpreted schemas. In *Proc. of the Ninth International Conference on Software Engineering*, 1987.
19. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.
20. Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2), June 1992.
21. Mark A. Linton. Encapsulating a C++ library. In *Proceedings of the 1992 USENIX C++ Conference*, pages 57–66, Portland, OR, August 1992.

22. Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.
23. William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA Conference Proceedings*, pages 145–161, Marist College, Poughkeepsie, NY, September 1990.
24. Charles Rich and Richard C. Waters. Formalizing reusable software components in the programmer's apprentice. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability, Volume II*, pages 313–343. Addison-Wesley, 1989.
25. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.
26. Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5), September 1984.
27. James C. Spohrer and Elliot Soloway. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, July 1992.
28. ParcPlace Systems. *ParcPlace Systems, Objectworks/Smalltalk Release 4 Users Guide.* Mountain View, California, 1990.
29. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 227–242, Orlando, Florida, October 1987.
30. John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.
31. André Weinand, Erich Gamma, and Rudolf Marty. ET++—An object-oriented application framework in C++. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 46–57, San Diego, CA, September 1988.
32. Rebecca Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

A Catalog Overview

The following summarizes the patterns in our current catalog.

Abstract Factory provides an interface for creating generic product objects. It removes dependencies on concrete product classes from clients that create product objects.

Adapter makes the protocol of one class conform to the protocol of another.

Bridge separates an abstraction from its implementation. The abstraction may vary its implementations transparently and dynamically.

Builder provides a generic interface for incrementally constructing aggregate objects. A Builder hides details of how objects in the aggregate are created, represented, and composed.

Command objectifies the request for a service. It decouples the creator of the request for a service from the executor of that service.

Composite treats multiple, recursively-composed objects as a single object.

Chain of Responsibility defines a hierarchy of objects, typically arranged from more specific to more general, having responsibility for handling a request.

Factory Method lets base classes create instances of subclass-dependent objects.

Flyweight defines how objects can be shared. Flyweights support object abstraction at the finest granularity.

Glue defines a single point of access to objects in a subsystem. It provides a higher level of encapsulation for objects in the subsystem.

Interpreter defines how to represent the grammar, abstract syntax tree, and interpreter for simple languages.

Iterator objectifies traversal algorithms over object structures.

Mediator decouples and manages the collaboration between objects.

Memento opaquely encapsulates a snapshot of the internal state of an object and is used to restore the object to its original state.

Observer enforces synchronization, coordination, and consistency constraints between objects.

Prototype creates new objects by cloning a prototypical instance. Prototypes permit clients to install and configure dynamically the instances of particular classes they need to instantiate.

Proxy acts as a convenient surrogate or placeholder for another object. Proxies can restrict, enhance, or alter an object's properties.

Solitaire defines a one-of-a-kind object that provides access to unique or well-known services and variables.

State lets an object change its behavior when its internal state changes, effectively changing its class.

Strategy objectifies an algorithm or behavior.

Template Method implements an abstract algorithm, deferring specific steps to subclass methods.

Walker centralizes operations on object structures in one class so that these operations can be changed independently of the classes defining the structure.

Wrapper attaches additional services, properties, or behavior to objects. Wrappers can be nested recursively to attach multiple properties to objects.

Abstract Factory

Object Creational

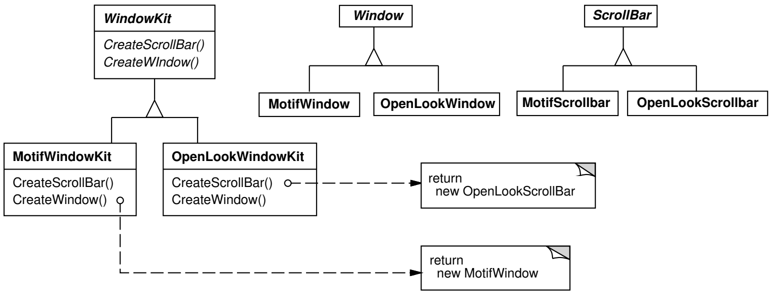
Intent

Abstract Factory provides an interface for creating generic product objects. It removes dependencies on concrete product classes from clients that create product objects.

Motivation

Consider a user interface toolkit that supports multiple standard look-and-feels, say, Motif and Open Look, and provides different scroll bars for each. It is undesirable to hard-code dependencies on either standard into the application—the choice of look-and-feel and hence scroll bar may be deferred until run-time. Specifying the class of scroll bar limits flexibility and reusability by forcing a commitment to a particular class instead of a particular protocol. An Abstract Factory avoids this commitment.

An abstract base class WindowKit declares services for creating scroll bars and other controls. Controls for Motif and Open Look are derived from common abstract classes. For each look-and-feel there is a concrete subclass of WindowKit that defines services to create the appropriate control. For example, the CreateScrollBar() operation on the MotifKit would instantiate and return a Motif scroll bar, while the corresponding operation on the OpenLookKit returns an Open Look scroll bar. Clients access a specific kit through the interface declared by the WindowKit class, and they access the controls created by a kit only by their generic interface.



Applicability

When the classes of the product objects are variable, and dependencies on these classes must be removed from a client application.

When variations on the creation, composition, or representation of aggregate objects or subsystems must be removed from a client application. Differences in configuration can be obtained by using different concrete factories. Clients do not explicitly create and configure the aggregate or subsystem but defer this responsibility to an AbstractFactory class. Clients instead call a method of the AbstractFactory that returns an object providing access to the aggregate or subsystem.

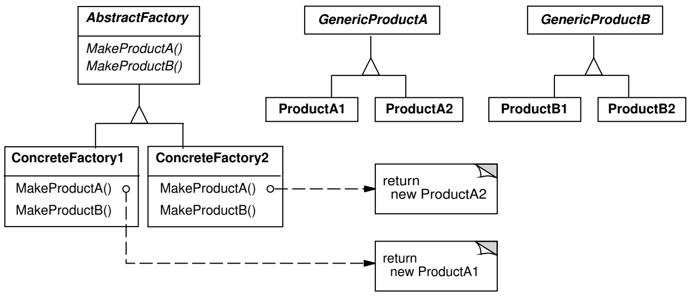
Participants

- **AbstractFactory**
  - declares a generic interface for operations that create generic product objects.
- **ConcreteFactory**
  - defines the operations that create specific product objects.
- **GenericProduct**
  - declares a generic interface for product objects.
- **SpecificProduct**
  - defines a product object created by the corresponding concrete factory.
  - all product classes must conform to the generic product interface.

Collaborations

- Usually a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To use different product objects, clients must be configured to use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclasses.

Diagram



Consequences

Abstract Factory provides a focus during development for changing and controlling the types of objects created by clients. Because a factory objectifies the responsibility for and the process of creating product objects, it isolates clients from implementation classes. Only generic interfaces are visible to clients. Implementation class names do not appear in client code. Clients can be defined and implemented solely in terms of protocols instead of classes.

Abstract factories that encode class names in operation signatures can be difficult to extend with new kinds of product objects. This can require redeclaring the AbstractFactory and all ConcreteFactories. Abstract factories can be composed with subordinate factory objects. Responsibility for creating objects is delegated to these sub-factories. Composition of abstract factories provides a simple way to extend the kinds of objects a factory is responsible for creating.

Examples

InterViews uses the "Kit" suffix [21] to denote abstract factory classes. It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects. InterViews also includes a LayoutKit that generates different composition objects depending on the layout desired.

ET++ [31] employs the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example). The WindowSystem abstract base class defines the interface for creating objects representing window system resources (for example, MakeWindow, MakeFont, MakeColor). Concrete subclasses implement the interfaces for a specific window system. At run-time ET++ creates an instance of a concrete WindowSystem subclass that creates system resource objects.

Implementation

A novel implementation is possible in Smalltalk. Because classes are first-class objects, it is not necessary to have distinct ConcreteFactory subclasses to create the variations in products. Instead, it is possible to store classes that create these products in variables inside a concrete factory. These classes create new instances on behalf of the concrete factory. This technique permits variation in product objects at finer levels of granularity than by using distinct concrete factories. Only the classes kept in variables need to be changed.

See Also

Factory Method: Abstract Factories are often implemented using Factory Methods.

Strategy

Object Behavioral

Intent

A Strategy objectifies an algorithm or behavior, allowing the algorithm or behavior to be varied independently of its clients.

Motivation

There are many algorithms for breaking a text stream into lines. It is impossible to hard-wire all such algorithms into the classes that require them. Different algorithms might be appropriate at different times.

One way to address this problem is by defining separate classes that encapsulate the different linebreaking algorithms. An algorithm objectified in this way is called a Strategy. InterViews [22] and ET++ [31] use this approach.

Suppose a Composition class is responsible for maintaining and updating the line breaks of text displayed in a text viewer. Linebreaking strategies are not implemented by the class Composition. Instead, they are implemented separately by subclasses of the Compositor class. Compositor subclasses implement different strategies as follows:

- **SimpleCompositor** implements a simple strategy that determines line breaks one at a time.
- **TeXCompositor** implements the $\TeX$algorithm for finding line breaks. This strategy tries to optimize line breaks globally, that is, one paragraph at a time.
- **ArrayCompositor** implements a strategy that is used not for text but for breaking a collection of icons into rows. It selects breaks so that each row has a fixed number of items.

A Composition maintains a reference to a Compositor object. Whenever a Composition is required to find line breaks, it forwards this responsibility to its current Compositor object. The client of Composition specifies which Compositor should be used by installing the corresponding Compositor into the Composition (see the diagram below).

Applicability

Whenever an algorithm or behavior should be selectable and replaceable at run-time, or when there exist variations in the implementation of the algorithm, reflecting different space-time tradeoffs, for example.

Use a Strategy whenever many related classes differ only in their behavior. Strategies provide a way to configure a single class with one of many behaviors.

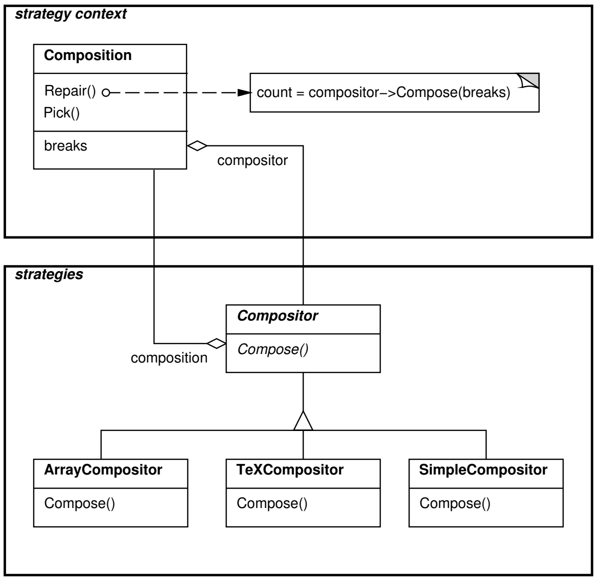
Participants

- **Strategy**
  - objectifies and encapsulates an algorithm or behavior.
- **StrategyContext**
  - maintains a reference to a Strategy object.
  - maintains the state manipulated by the Strategy.
  - can be configured by passing it an appropriate Strategy object.

Collaborations

- Strategy manipulates the StrategyContext. The StrategyContext normally passes itself as an argument to the Strategy's methods. This allows the Strategy to call back the StrategyContext as required.
- StrategyContext forwards requests from its clients to the Strategy. Usually clients pass Strategy objects to the StrategyContext. Thereafter clients only interact with the StrategyContext. There is often a family of Strategy classes from which a client can choose.

Diagram



Consequences

Strategies can define a family of policies that a StrategyContext can reuse. Separating a Strategy from its context increases reusability, because the Strategy may vary independently from the StrategyContext.

Variations on an algorithm can also be implemented with inheritance, that is, with an abstract class and subclasses that implement different behaviors. However, this hard-wires the implementation into a specific class; it is not possible to change behaviors dynamically. This results in many related classes that differ only in some behavior. It is often better to break out the variations of behavior into their own classes. The Strategy pattern thus increases modularity by localizing complex behavior. The typical alternative is to scatter conditional statements throughout the code that select the behavior to be performed.

Implementation

The interface of a Strategy and the common functionality among Strategies is often factored out in an abstract class. Strategies should avoid maintaining state across invocations so that they can be used repeatedly and in multiple contexts.

Examples

In the RTL System for compiler code optimization [17], Strategies define different register allocation schemes (RegisterAllocator) and different instruction set scheduling policies (RISCscheduler, CISCscheduler). This gives flexibility in targeting the optimizer for different machine architectures.

The ET++SwapsManager calculation engine framework [10] computes prices for different financial instruments. Its key abstractions are Instrument and YieldCurve. Different instruments are implemented as subclasses of Instrument. The YieldCurve calculates discount factors to present value future cash flows. Both of these classes delegate some behavior to Strategy objects. The framework provides a family of Strategy classes that define algorithms to generate cash flows, to value swaps, and to calculate discount factors. New calculation engines are created by parameterizing Instrument and YieldCurve with appropriate Strategy objects. This approach supports mixing and matching existing Strategy implementations while permitting the definition of new Strategy objects.

See Also

Walker often implements algorithms over recursive object structures. Walkers can be considered compound strategies.

Intent

A Wrapper attaches additional services, properties, or behavior to objects. Wrappers can be nested recursively to attach multiple properties to objects.

Motivation

Sometimes it is desirable to attach properties to individual objects instead of classes. In a graphical user interface toolkit, for example, properties such as borders or services like scrolling should be freely attachable to any user interface component.

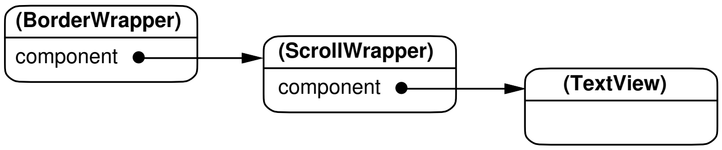
One way to attach properties to components is via inheritance. Inheriting a border from a base class will give all instances of its derived classes a border. This is inflexible because the choice of border is made statically. It is more flexible to let a client decide how and when to decorate the component with a border.

This can be achieved by enclosing the component in another object that adds the border. The enclosing object, which must be transparent to clients of the component, is called a Wrapper. This transparency is the key for nesting Wrappers recursively to construct more complex user interface components. A Wrapper forwards requests to its enclosed user interface component. The Wrapper may perform additional actions before or after forwarding the request, such as drawing a border around a user interface component.

Typical properties or services provided by user interface Wrappers are:

- decorations like borders, shadows, or scroll bars; or
- services like scrolling or zooming.

The following diagram illustrates the composition of a TextView with a BorderWrapper and a ScrollWrapper to produce a bordered, scrollable TextView.



Applicability

When properties or behaviors should be attachable to individual objects dynamically and transparently.

When there is a need to extend classes in an inheritance hierarchy. Rather than modifying their base class, instances are enclosed in a Wrapper that adds the additional behavior and properties. Wrappers thus provide an alternative to extending the base class without requiring its modification. This is of particular concern when the base class comes from a class library that cannot be modified.

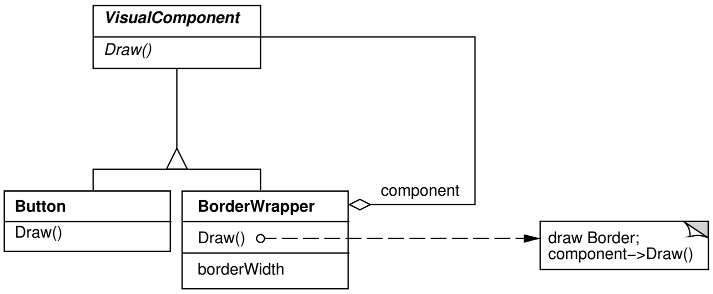
Participants

- **Component**
  - the object to which additional properties or behaviors are attached.
- **Wrapper**
  - encapsulates and enhances its Component. It defines an interface that conforms to its Component's.
  - Wrapper maintains a reference to its Component.

Collaborations

- Wrapper forwards requests to its Component. It may optionally perform additional operations before and after forwarding the request.

Diagram



Consequences

Using Wrappers to add properties is more flexible than using inheritance. With Wrappers, properties can be attached and detached at run-time simply by changing the Wrapper. Inheritance would require creating a new class for each property composition (for example, BorderdScrollableTextView, BorderedTextView). This clutters the name space of classes unnecessarily and should be avoided. Moreover, providing different Wrapper classes for a specific Component class allows mixing and matching behaviors and properties.

Examples

Most object-oriented user interface toolkits use Wrappers to add graphical embellishments to widgets. Examples include InterViews [22], ET++ [31], and the ParcPlace Smalltalk class library [28]. More exotic applications of Wrappers are the DebuggingGlyph from InterViews and the PassivityWrapper from ParcPlace Smalltalk. A DebuggingGlyph prints out debugging information before and after it forwards a layout request to its enclosed object. This trace information can be used to analyze and debug the layout behavior of objects in a complex object composition. The PassivityWrapper can enable or disable user interactions with the enclosed object.

Implementation

Implementation of a set of Wrapper classes is simplified by an abstract base class, which forwards all requests to its component. Derived classes can then override only those operations for which they want to add behavior. The abstract base class ensures that all other requests are passed automatically to the Component.

See Also

Adapter: A Wrapper is different from an Adapter, because a Wrapper only changes an object's properties and not its interface; an Adapter will give an object a completely new interface.

Composite: A Wrapper can be considered a degenerate Composite with only one component. However, a Wrapper adds additional services—it is not intented for object aggregation.

This article was processed using the LaTeX macro package with LLNCS style