Relationships between Design Patterns

Walter Zimmer
Forschungszentrum Informatik, Bereich Programmstrukturen
Haid-und-Neu-Strasse 10-14, D-76131 Karlsruhe, Germany
email: zimmer@fzi.de

Abstract

The catalogue of design patterns from [Gamm94] contains about twenty design patterns and their mutual relationships. In this paper, we organize these relationships into different categories and revise the design patterns and their relationships. We are then able to arrange the design patterns in different layers. The results simplify the understanding of the overall structure of the catalogue, thereby making it easier to classify other design patterns, and to apply these design patterns to software development.

1. Introduction

In the last couple of years, object-orientation has gained much attention in the field of software engineering. However, after some initial experiences with object-orientation, software engineers are facing fundamental problems when designing and reusing applications and libraries, which cannot be solved by current methods and tools.

A growing number of people consider design patterns to be a promising approach to system development, which addresses the aforementioned problems, especially in object-oriented systems (cf. [Beck93], [Beck94], [Gamm93], [Gamm94], [Coad93a], [Copl91], [Copl94], [Shaw91], [John92], [Busc93], [Pree94]). The main idea behind design patterns is to support the reuse of design information, thus allowing developers to communicate more effectively. New design patterns are being discovered, described and applied by several research groups. Development tools supporting the design pattern approach are also under work.

In this new field of software engineering, the following important questions are arising:

- Amongst the many different design patterns that are being discovered, are any related to each other? What are the characteristics of such a relationship?
- Do two patterns address a similar problem area?
- Is it possible to combine two design patterns?
- What are the criteria for classifying design patterns into categories?

Very few publications [Gamm93] [Gamm94] [1] have adequately addressed these issues. Similar to the pattern descriptions in architecture [Alex77], each pattern description in [Gamm94] contains a "See Also" section with possible relationships between design patterns. Furthermore, the catalogue presents a classification of all design patterns according to two criteria: jurisdiction (class, object, compound) and characterization (creational, structural, behavioural).

The relationships in [Gamm94] are described informally and in detail, so that each relationship appears to be a little bit different from the other ones. We propose a classification of the relationships which helps in understanding the similarities among the relationships. This motivates us to modify the catalogue slightly, and to organize the design patterns into several layers.

This paper gives new insights into the relationships between existing design patterns. Our major accomplishments are as follows:

- a classification of the relationships between design patterns;
- a new design pattern resulting from a generalization of several other design patterns;
- and a structuring of design patterns into several layers.

In the next chapter, we present a graphic view of all design patterns and their relationships as they appear in the aforementioned catalogue [Gamm94]. Section 3 classifies these relationships. This process raises some problems and gives further insights into the relationships between design patterns in general. After modifying the catalogue structure in Section 4, we show in Section 5 how it is possible to ar-

1. In this paper we will refer to the preliminary version of this catalogue dated from 9/93. We suppose that the reader has at least some knowledge about the design patterns of this catalogue. [Gamm93] is a good introductory paper for design patterns and this catalogue. In [Gamm94], the authors have updated their terminology: Cookie → Memento, Exemplar → Prototype, Manager → Mediator, Mimic → State, Walker → Visitor, Wrapper → Decorator. Descriptions of almost all design patterns ofthe forthcoming catalogue can be found on *st.cs.uiuc.edu* under */pub/patterns/dpcat.*

range the design patterns into layers representing different abstraction levels.

2. Overall structure of the design pattern catalogue

Figure 1 is a graphic presentation of the design patterns and their relationships that were in [Gamm94]. No further information is added to this figure. The annotations to the arrows are taken almost literally from this catalogue. The variables X and Y are placeholders for the source and target of the respective arrows.

Figure 1 gives an impression of the overall structure of the catalogue. It serves as a reference point for the rest of this paper, because it contains the most detailed information about the relationships. Figure 1 serves therefore as the starting point for the further classification and revision of the relationships.

3. Classification of relationships

3.1 Issues to be addressed

On the basis of Figure 1, we tried to find categories containing similar kinds of design pattern relationships. During this process, the following issues had to be addressed.

Relationships refer to different aspects of design patterns

Figure 1 shows relationships addressing issues ranging from the problem definition ("creating objects", "decoupling of objects") and the solution definition ("Command can use the Composite pattern in its solution") to very specific implementation details ("similar in a level of indirection").

We wanted to improve the comprehensibility of the catalogue and its structure. Therefore, we focused on the

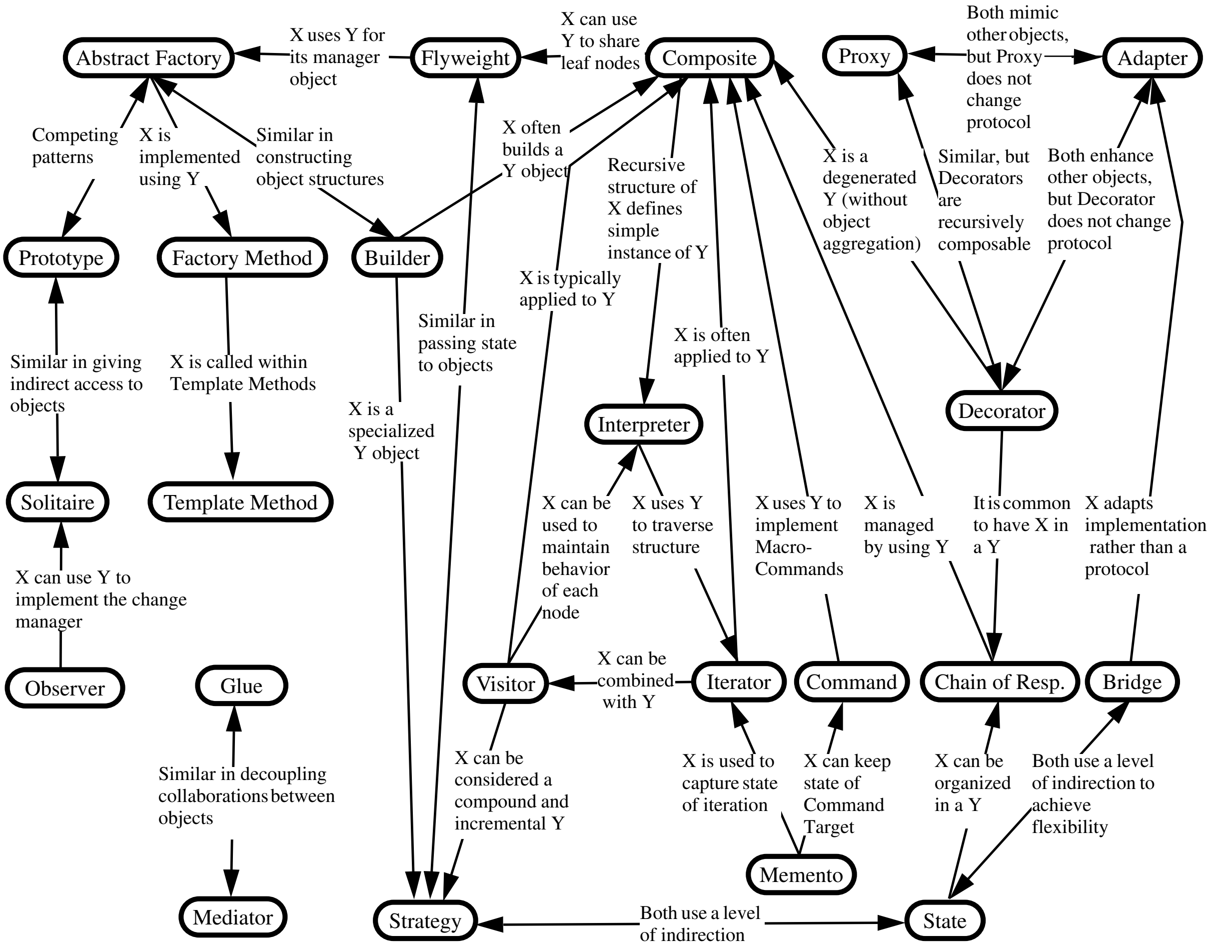


Figure 1 The starting point: The overall structure of the design pattern catalogue

problem and solution aspects. The relationships addressing implementation details played a minor role in this paper. For this reason, they were removed in later figures (Figure 3 and Figure 4).

Directions of relationships

The descriptions of the relationships in the catalogue do not contain the direction of the relationships, which may be unidirectional — backwards or forwards — or bidirectional. Once the category that the relationship belongs to, is known, the direction can be determined rather quickly. But if one is not sure about the category, then the direction often remains unclear.

Strength of relationship

Is the given relationship quite strong (like between Abstract Factory and Factory Method) or are the design patterns more loosely coupled (like Observer and Solitaire)? Some design patterns can stand almost alone, others make sense only in combination with certain other design patterns. The assessment of this property is often subjective because it is based on experience in using and combining different design patterns.

3.2 Categories of relationships

We have classified relationships between the pairs (X,Y) of design patterns into the following categories:

X uses Y in its solution

When building a solution for the problem addressed by X, one subproblem is similar to the problem addressed by Y. Therefore, the design pattern X uses the design pattern Y in its solution. Thus, the solution of Y (e.g. class structures) represents one part of the solution of X.

X is similar to Y

They address a similar kind of problem (not a similar kind of solution). In several cases, these similarities are also expressed in the classification given in the catalogue, e.g. the catalogue classifies both Prototype and Abstract Factory in the category “Object - Creational”.

X can be combined with Y

A typical combination of design patterns is the combination of X and Y (e.g. Iterators traverse Composite structures). In contrast to “X uses Y”, X does not use Y in its solution (or vice versa).

3.3 Classification

Figure 2 shows how we have classified the relationships depicted in Figure 1 into these categories. Instead of justifying the classifications of all relationships, we focus on the most interesting ones. Interesting means here, that we find it difficult to assign a particular relationship into exactly one of the categories, that we are not sure about the meaning of the relationship at all, or we think that some further explanation is useful.

X uses Y in its solution

In most cases, the assignment of relationships to this category is clear. Perhaps it would make sense to split these relationships further into “X must use Y” and “X might use Y”, because this information indicates the strength of the relationship.

X is similar to Y

Abstract Factory, Prototype and Builder are similar in that they all deal with object creation. Builder and Visitor are strategies. Both Glue and Mediator serve to decouple objects. In contrast to Proxy and Decorator which allow the client to attach additional properties dynamically to an object, Adapter primarily serves to provide a completely different interface to an object.

State (Strategy) is rather loosely coupled with Bridge and Strategy (Flyweight) as this relationship addresses the implementation detail “level of indirection” (“passing of state to objects”).

X can be combined with Y

A Builder often produces Composite objects. A Factory Method is typically called in a Template Method.

Composite, Visitor, Iterator: Iterator traverses composite structures and Visitor centralizes operations on object structures. Depending upon the necessary degree of flexibility, one typically combines two or all three design patterns (for instance Interpreter).

Composite, Decorator: Composite and Decorator are often used together in applications, for example for visual objects in ET++, MacApp and Interviews [Wein88], [App89], [Lint89]. There are also other kinds of relationships between them: when looking at the solution aspect, Decorator can be seen as a degenerated Composite; when considering the problem aspect, they both support recursively structured objects, whereby Decorator focuses on attaching additional properties to objects. Thus, the design patterns are somehow similar, but it difficult to state it more precisely. Therefore, we only insert a relationship of type “combined”, and neglect the other ones.

3.4 Using the classification

This section gives several possibilities for using the presented classification when working with design patterns.

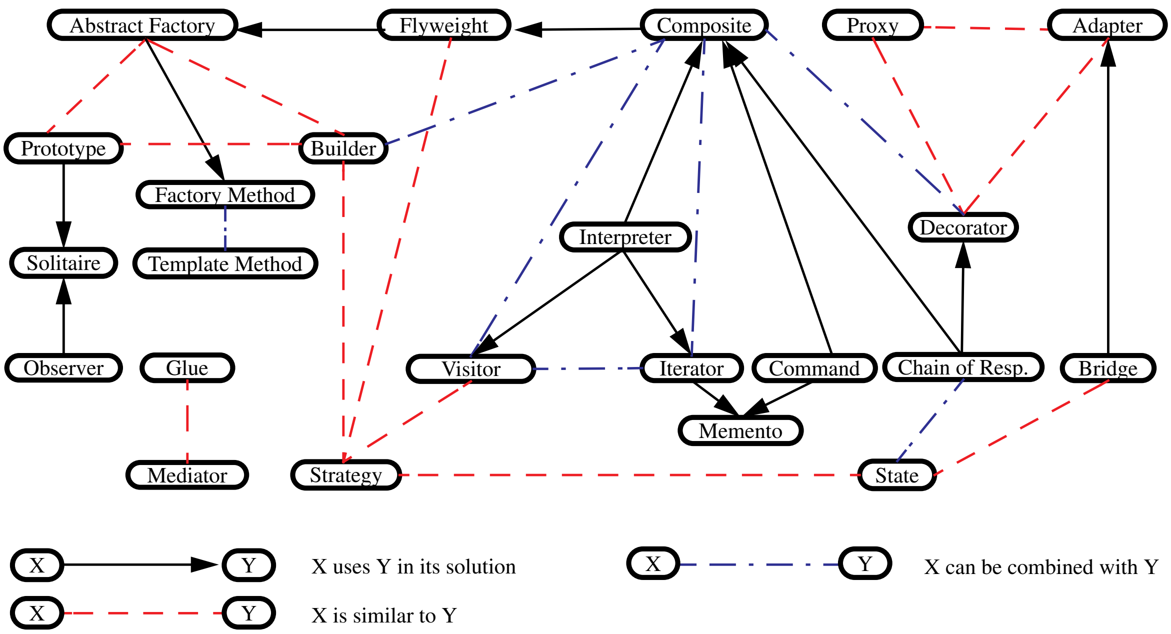


Figure 2 Classification of Relationships

X uses Y in its solution

This relationship makes clear, that Y can be used as a part of the solution of X. The description of X can refer back to that of Y, in order to be shorter and easier to understand. Tools supporting the design pattern approach can profit from this information; the relationship can be checked in existing designs; design patterns like Y can be visualized as blocks without internal implementation details in order to raise the abstraction level.

X is similar to Y

Design patterns related in this way address similar problems. When searching for a design pattern which solves a certain kind of problem, first of all you can look at a family of similar design patterns addressing this kind of problems; secondly, you use the one which meets your requirements best. Thus, this relationship supports the retrieval of design patterns.

X can be combined with Y

When one has already applied a design pattern in a system, relationships of this kind can help in finding other design patterns which are useful to combine with the existing one. Thereby, the retrieval of design patterns is supported.

Several design patterns related by this relationship may also be used as larger building blocks in design, thus raising the abstraction level.

4. Modifying relationships and design patterns

This section examines the proposed classification of the relationships further. This process results in a new design pattern and some modifications to existing relationships.

4.1 A new design pattern Objectifier

Figure 2 shows that Strategy is similar to the design patterns Builder and Visitor which can be regarded as special kinds of strategies objectifying some behaviour. Iterator and Command are also design patterns which objectify certain behaviours; the catalogue writes: “a Command objectifies command dependent behaviour” or “an Iterator allows to vary the traversal of object structures”.

We consider the objectification of behaviour, by means of additional classes, to be the central common factor of these and other design patterns. The usage of this design pattern allows to vary this objectified behaviour.

Therefore, we think that the objectification of behaviour is a basic design pattern; we call it *Objectifier*. A detailed description of Objectifier is given in the appendix. It uses the same description format as [Gamm93]. The Implementation and Sample Code parts are left out.

4.2 Other modifications

The organization of the relationships in different categories is sometimes difficult, because it partly depends upon subjective criteria. The difference between “X may use Y” or “X can be combined with Y” depends upon a subjective assessment, whether the usage of Y is seen as a central part of the solution of X, or if it is more a combination of two autonomous design patterns.

Furthermore, two design patterns might be related in different ways; Decorator / Composite and Abstract Factory / Prototype are pairs of design patterns, which can be combined, and are also similar. In our paper, each relationship is assigned to the most adequate category.

Factory Method

Figure 2 shows that Factory Method does not *use* Template Method, but it is often *called* in a Template Method, i.e. Factory Method often plays the role of a primitive in a Template Method. Thus, if an Abstract Factory uses Factory Method in its solution, then it really uses the design pattern Template Method. Therefore, we do not consider Factory Method as a real design pattern, but as a “X use Y” relationship between Abstract Factory and Template Method. Figure 3 draws an “X use Y” arrow between these design patterns and annotates it with Factory Method.

Adapter — Decorator, Proxy

Among other things the catalogue says “Decorator is different from an Adapter, because a Decorator only changes an object‘s properties and not its interface; an Adapter will give an object a completely new interface.” We think that this is a main difference, and that Adapter is quite different from both Decorator and Proxy. This relationship is therefore removed.

Flyweight — Abstract Factory

As the manager part of Flyweight is no intrinsic part of the design pattern, we remove this relationship.

Objectifier — Flyweight, State

The relationships from Strategy to other design patterns are transformed in corresponding relationships to Objectifier. As we neglect relationships addressing implementation details, we remove these relationships from Objectifier to Flyweight and State. But as explained in the previous section about Objectifier, State uses Objectifier in its own solution. Therefore, a new relationship between State and Objectifier is added.

Adapter — Bridge

Bridge may use Adapter in its own solution. An example for this is the data structure set: Adapters can be used to view lists, array and tables as sets. Thus Adapters standardize the interfaces of the different ConcreteImplementor classes (lists, array and tables) to the common Implementor interface in the Bridge pattern.

Objectifier — Template Method

As explained in description of Objectifier (see Appendix), both patterns serve the similar purpose of varying behaviour.

The integration of all these modifications, as well as the addition of the Objectifier pattern, into Figure 2 results in Figure 3.

5. Layers of design patterns

5.1 Arrangement in several layers

Up to now, we have classified the relationships between the design patterns and modified some of them. As one can see in Figure 3, “X uses Y” is the most frequent relationship.

We therefore try to arrange the patters according to this predominant relationship. The graph defined by the “X uses Y” relationship is acyclic. This property allows us to arrange the design patterns straightforwardly in different layers as shown in Figure 4.

Thus we identify three semantically different layers:

- Basic design patterns and techniques.
- Design patterns for typical software problems.
- Design patterns specific to an application domain.

5.2 Basic design patterns and techniques

This layer contains the design patterns, which are heavily used in the design patterns of higher layers and in object-oriented systems in general. The two design patterns Objectifier and Composite seem to be the most important ones as they are used by eight, respectively three other design patterns.

The problems addressed by these design patterns occur again and again when developing object-oriented systems. The design patterns are thus very general. When building a system, one would often look upon them more as basic design techniques than as patterns.The intentions of these design patterns (see Table 1) are very general and applicable to a broad range of problems occurring in the design of object-oriented systems.

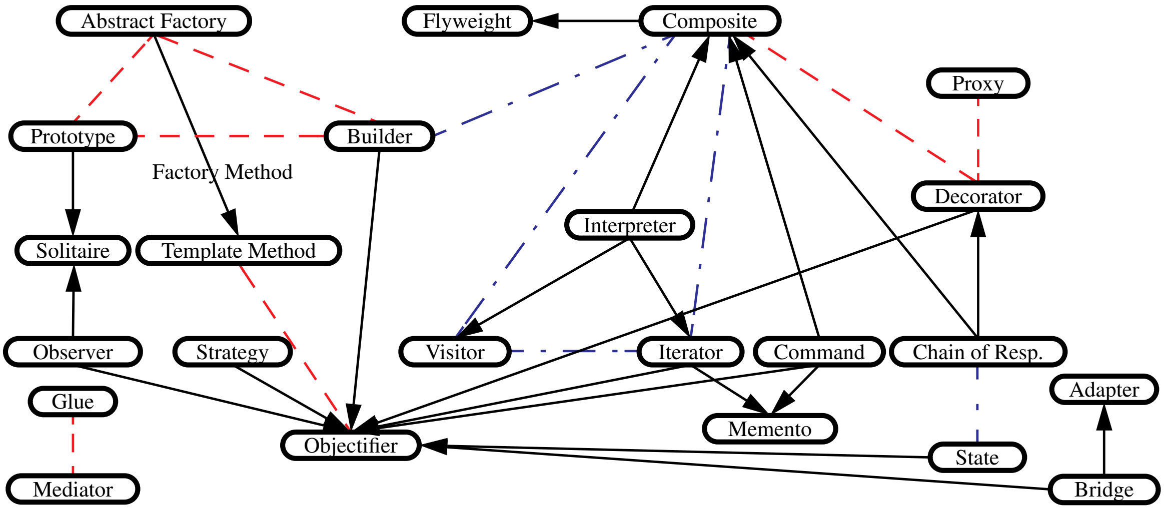


Figure 3 Revised Classification

Design pattern	Purpose of the design pattern
Adapter	Adapting a protocol of one class to the protocol of another class.
Composite	Single and multiple, recursively composed objects can be accessed by the same protocol.
Decorator	Attaching additional properties to objects.
Glue	Encapsulating a subsystem.
Mediator	Managing collaboration between objects.
Memento	Encapsulating a snapshot of the internal state of an object.
Objectifier	Objectifying behaviour.
Proxy	Controlling access to an object.
Solitaire	Providing unique access to services or variables.
Template Method	Objectifying behaviour (primitives will be varied in subclasses).

Table 1 Basic design patterns with their respective purposes

We regard the problem addressed by Composite (and Decorator, Proxy) as a bit more specific than the problems addressed by the other patters of this layer. Therefore, we thought about moving Composite into the next higher layer. But Composite is a basic design pattern, in the sense that, the addressed problem of handling recursively structured objects is a basic problem in many contexts. The many relationships to Composite in Figure 4 express this quite well. We therefore leave Composite in the basic layer.

5.3 Design patterns for typical software problems

The middle layer comprises design patterns which are used for more specific problems in the design of software. These design patterns are not used in design patterns from the basic layer, but in patterns from the application specific layer, and possibly from the same layer. The problems addressed by these design patterns are not typical of a certain application domain.

Builder, Prototype and Abstract Factory address problems with the creation of objects; Iterator traverses object structures; Command objectifies an operation; and so on.

5.4 Design patterns specific to application domain

Design patterns in this layer are the most specific and they can often be assigned to one or more application domains.

Although the general problem of parsing some input often occurs, we consider Interpreter to be more specific. Interpreter is used to parse simple languages. The catalogue lists some of the known uses of Interpreter, e.g. parsing constraints and matching regular expressions. Compiler construction is the major application domain.

The current catalogue contains almost no application specific design patterns. Most patterns are generic and ap-

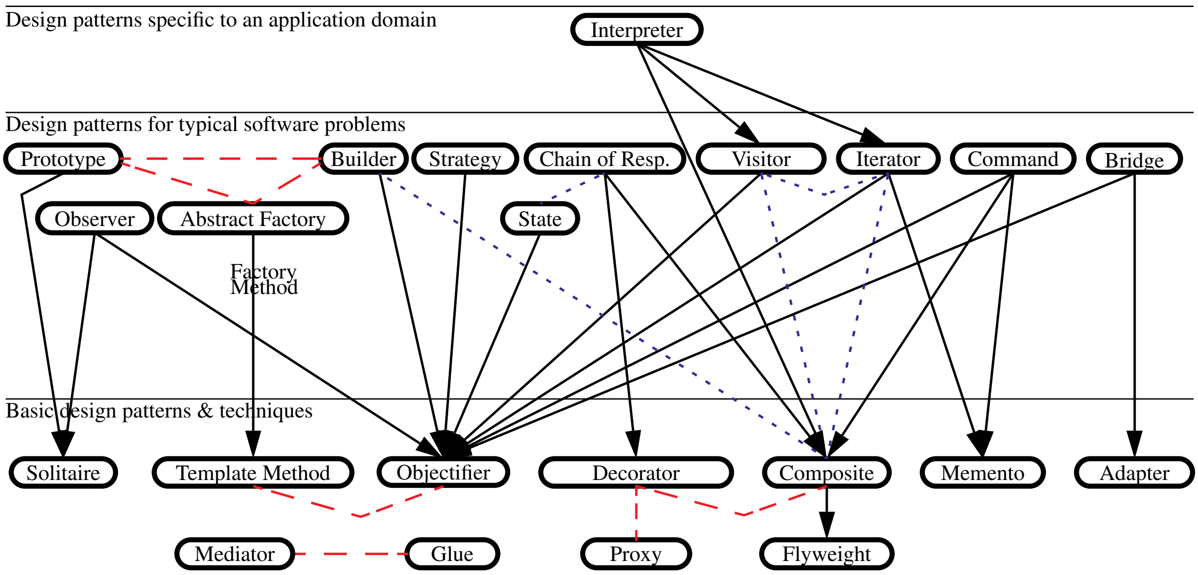


Figure 4 Arrangement of design pattern in layers

plicable to a broad range of problems. The authors support this property by the existence of a "Known Uses" section in the design pattern description, which should include at least two examples from different application domains.

5.5 Other arrangements

The presented arrangement of design patterns into layers is only one possible separation of design patterns. We see it as a starting point for further work. As more design patterns are described, the separation in layers should be discussed and adapted to new requirements. At the moment, it helps us to grasp and to understand the overall structure of the catalogue, and to relate new design patterns to existing ones. It is also an aid for traversing / learning design patterns, as the user can choose between a bottom-up or a top-down traversal.

Another possible arrangement is to group design patterns according to their typical combinations of design patterns. Up to now, only some such combinations are known (see Figure 4). We think, that in the future they will play a more important role, because typical combinations can be used as building blocks in design.

The criteria jurisdiction and characterization, which are given in [Gamm94], result in several clusters of design patterns with a similar intent; this arrangement can help during the retrieval of an adequate design pattern for a specific problem.

6. Related Work

The notion of design patterns is introduced by [Alex77] in the area of architecture. Each design pattern description contains a section where relationships to other patterns, of the same, of a higher or of a lower granularity level are presented. These relationships influence the construction process, because one should always look at related patterns when one builds something; and because one should always apply patterns of higher levels first. A classification for the patterns, but not for their mutual relationships, is given.

[Gamm93] [Gamm94] present a large collection of well described design patterns. The relationships between design patterns are also described, but not classified, although a classification of design patterns is included. Their clustering according to jurisdiction (class, object, compound) and characterization (creational, structural, behavioural) is orthogonal to the one derived in this paper. Patterns in a certain cluster can be considered as similar to one another, thereby supporting the selection of an appropriate design pattern for a certain problem.

Frameworks [WB90] [John91] can be considered as high-level design patterns, usually consisting of many interrelated design patterns of lower levels. In [Beck94] the authors write "Patterns can be used at many levels, and what is derived at one level can be considered a basic pattern at another level." Furthermore, they state "This is probably typical of most architectures; some patterns will be generic and some will be specific to the application domain", which confirms the organization depicted in Figure

4. [Booc93] also mentions that design patterns are ranging from idioms to frameworks.

In [Coad93b], several design patterns are combined in an exemplary application, but the relationships are not investigated any further.

7. Conclusion

We have presented a classification of the relationships between design patterns, which led to a new design pattern and to an arrangement of the design patterns into different layers. These results partially stem from one of our former projects [Zimm94]. Although the design pattern approach and the excellent catalogue [Gamm94] have proven effective in this project, the following issues showed up:

- The design of important abstractions of the application domain often requires the combination of several, interrelated design patterns.
- Applying design patterns requires a fair knowledge of both single design patterns and their relationships.
- Tool support is needed to apply design patterns to really large applications.

Our results address these issues because they help in:

- understanding the often complex relationships between design patterns;
- organizing existing design patterns, as well as categorizing and describing new design patterns;
- comparing different collections of design patterns;
- and in building CASE tools which support design patterns.

Thus, the results are a step towards the development of a pattern language. We think that this will be very valuable for the development of object-oriented systems.

We are continuing our work with design patterns by formalizing the semantics of the different kinds of relationships and the different layers shown in Figure 4. We are aiming at greater precision and a better semantic definition. This is a prerequisite for defining a generally accepted and usable classification scheme, which will serve as a basis for further work.

Many design patterns are being discovered and described outside of those in [Gamm94], especially application specific design patterns. We will organize them and their relations in the given classification scheme. This will enable us to evaluate the validity and usefulness of the classification scheme and improve it accordingly.

8. References

[Alex77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[App89] Apple Computer, Cupertino, California. *Inc. Macintosh Programmers Workshop Pascal 3.0 Reference*, 1989.

[Beck93] K. Beck. Patterns and software development. *Dr. Dobbs Journal*, 19(2):18–23, 1993.

[Beck94] K. Beck and R. Johnson. Patterns generate architecture. In *Proceedings of ECOOP'94*, 1994. To appear.

[Booc93] G. Booch. Patterns. *Object Magazine*, 3(2), 1993.

[Busc93] F. Buschmann. Rational architectures for object-oriented software systems. *Journal of Object-Oriented Programming*, 6(5):30–41, September 1993.

[Casa92] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietmar Theobald, and Walter Zimmer. Obst - an overview. Technical report, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, June 1992. FZI.039.1.

[Coad93a] P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):153–159, September 1993.

[Coad93b] P. Coad. Patterns (workshop). In *OOPSLA'92 Addendum to the Proceedings*, volume 4 of *OOPS Messenger*, pages 93–96, Vancouver, B.C., Canada, October 1993. OOPS Messenger, ACM Press.

[Copl91] J. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1991.

[Copl94] J.O. Coplien. Generative pattern languages: An emerging direction of software design. Technical report, 1994.

[Gamm93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, pages 406–431, Berlin, 1993. Springer-Verlag.

[Gamm94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Pattern*. Addison-Wesley, To Appear, 1994.

[John91] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report Technical Report UIUCDCS 91–1696, University of Illinois, May, 1991.

[John92] R. Johnson. Documenting frameworks using patterns. In *Proceedings of OOPSLA'92*, volume 27 of *ACM SIGPLAN Notices*, pages 63–76, Vancouver, B.C., Canada, October 1992. ACM Press.

[Lint89] M. Linton, John Vlissides, and P. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, February 1989.

[Pree94] W. Pree. Meta-patterns: A means for describing the essentials of reusable o-o design. In *Proceedings of ECOOP'94*, 1994. To Appear.

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F.Eddy, and

W.Lorensen. *Object-Oriented Modeling And Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Shaw91] M. Shaw. Heterogenous design idioms for software architecture. In *Proceeding of the Sixth International Workshop on Software Specification and Design*, Software Engineering Notes, pages 158–165, Como, Italy, October 25-26 1991. IEEE Computer Society.

[WB90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *CACM*, 33(9):105–123, September 1990.

[Wein88] André Weinand, Erich Gamma, and Rudolph Marty. ET++ – an object-oriented application framework in C++. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, pages 46–57, November 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[Zimm94] Walter Zimmer. Experiences using design patterns to reorganize an object-oriented application, July 1994. Position paper for the Pattern Workshop at ECOOP'94.

A Description of Objectifier

Name

Objectifier

Intent

Objectify similar behaviour in additional classes, so that clients can vary such behaviour independently from other behaviour, thus supporting variation-oriented design (see [Gamm93]). Instances from those classes represent behaviour or properties, but not concrete objects from the real world.

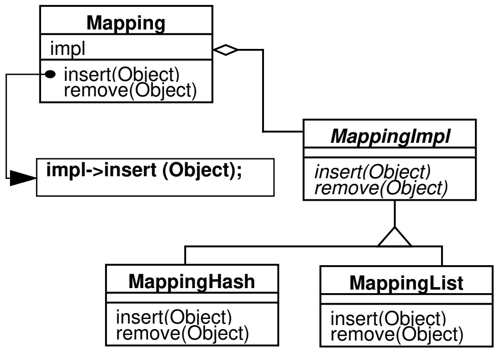
Motivation

Objectifier is a very general design pattern to be applied in a wide range of problems. The idea of objectifying behaviour is used in a lot of other design patterns. In the following, the usage of Objectifier is shown with an example, which represents a somewhat simpler variant of the design pattern Bridge.

A frequent problem in design is the separation of an abstraction from its implementation, and the interchange of implementations. For example, a data type Mapping might provide different implementations for different tasks due to reasons of efficiency.

A common approach is to have an abstract class for the data type Mapping with concrete subclasses MappingList and MappingHash representing different implementations. This allows to interchange the implementation at compile-time, but *not* at run-time.

A more flexible approach is to objectify the varying behaviour, i.e. to have independent implementation objects which can be interchanged at run-time. In the example of Mapping[1], you have a class Mapping representing the abstraction, and an abstract class MappingImpl, which is the superclass for the concrete implementation classes MappingList, MappingHash. Mapping maintains a reference to MappingImpl, and it delegates the requests to its current implementation object.



This solution allows to interchange the implementation object at run-time.

Applicability

Use the Objectifier pattern when

- Behaviour should be decoupled from classes in order to have independent behaviour objects which can be interchanged, saved, modified, shared or invoked.
- Run-time configuration of behaviour is required.
- There are several almost identical classes which differ only in one or a few methods. Objectifying the different behaviour in additional classes allows to unify the former classes in one common class, which can then be configured with a reference to the new, additional classes.
- There is a large amount of conditional code to select behaviour.

Participants

Client

- has a reference to the Objectifier.
- can be configured with a concrete Objectifier at run-time.

Objectifier

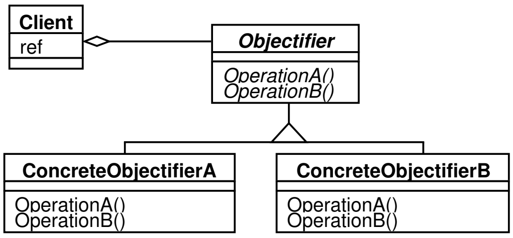
1. The OMT Notation [Rumb91] is used in this diagram. Italic letters indicate abstract classes and methods. The method insert from Mapping is represented in pseudo-code. The attribute impl of Mapping is a reference to MappingImpl.

- defines the common interface for the different concrete Objectifiers. It may also contain data or references to other objects which are common to all concrete Objectifiers.

Collaborations

- A client may use Objectifier to delegate parts of its behaviour. The Objectifier receives the information needed to fulfil its task during its initialization, or the client passes the information as a parameter when calling the Objectifier.
- A client can be configured with a concrete Objectifier to adapt the behaviour to a current situation.

Class Diagram



Consequences

- Encapsulation, Modularity: Behaviour is objectified and encapsulated in classes.
- Configurability, Customizability: Clients of Objectifier can change the concrete Objectifier at run-time.
- Extensibility, Single Point of Evolution: New behaviour can be implemented by adding a new class without affecting existing classes.
- Efficiency
  - — Loss of time and space by an additional level of indirection.
  - — The client can dynamically select the class which is most efficient according to the current situation.
  - — Stateless Objectifiers (without attributes) can be shared by different objects.

Known Uses

The example of the data type Mapping (see Motivation section) is taken from the data structure library of OBST, an object-oriented database system [Casa92]. The type of the current implementation object of a Mapping object depends upon the current size; if the number of objects managed by the Mapping object exceeds eight (or falls below four), then the list implementation is replaced with a hash implementation (or vice versa). This replacement is triggered by the Mapping object itself, not by the user.

Objectifier is also used in the solution of other design patterns. Therefore, one can find real examples of Objectifier by looking at the design patterns referenced in the "See Also" section.

See Also

Although several design patterns contain the common idea of objectifying behaviour to solve a problem, their purposes and requirements are more specific than those of Objectifier. Therefore, Objectifier is rather a generalization of these patterns than a totally new design pattern. It removes redundancies in the descriptions of other design patterns

As a lot of other issues have to be addressed when applying these related design patterns, we regard them as independent design patterns, not only as specialized variants of Objectifier. Table 2 contains related design patterns with the corresponding behaviour to be objectified (and potentially varied).

Design pattern	Objectified behaviour
Bridge	Implementation of some abstraction
Builder	Creation / Representation of objects
Command	Command dependent behaviour
Iterator	Traversal of object structures
Observer	Context dependent behaviour
State	State dependent behaviour
Strategy	(Complex) Algorithm
Visitor	Type dependent behaviour (types of single objects in Compound structure)

Table 2 Design patterns with their objectified behaviour

Strategy: Objectifier differs from Strategy in objectifying behaviour in a broader sense and is not restricted to algorithms in the classic sense of "algorithms and data structures". Thus, Objectifier is more general than Strategy.

Template Method: It has a similar intent as Objectifier: the variation of some behaviour. A Template Method represents the principal structure of an algorithm or behaviour, whereby parts of it can be varied in subclasses by (re)defining methods (primitives). In contrast to this, Objectifier puts only the variable parts in additional classes, so that these parts can be varied independently from other behaviour. One superclass defines the common interface, and several subclasses implement the concrete behaviour in different ways.