HACKING THE LIBRARY

BOOK CLUB

Patterns

Here is a framework to add structure and depth to what is likely a way of thinking you regularly employ.

Please skim the attached documents.

About Patterns and Pattern Languages

Patterns are the recurring solutions to the problems of design. People learn patterns by seeing them and recall them when need be without a lot of effort. Patterns link together in the mind so that one pattern leads to another and another until familiar problems are solved. That is, patterns form languages, not unlike natural languages, within which the human mind can assemble correct and infinitely varied statements from a small number of elements.

Ward Cunningham,
The Portland Pattern Repository

A Pattern Language
Christopher Alexander

Software Patterns
Selected Excerpts

DUE MONDAY, JAN. 30

R FOR ENVIRONMENTAL STRUCTURE SERIES

The Timeless Way of Building, lays the foundation
. It presents a new theory of architecture, build-
nning which forms the basis for a new traditional
ial architecture, created by the people . .

inspiration; as a practicing planner, an educator,
you cannot help but be challenged and stimulated
."

Dennis Michael Ryan, *Journal of the American Planning Association*

l *Pattern Language*, is a working document for
tecture. It is an archetypal language which allows
o design for themselves.

ıis to be perhaps the most important book on
design published this century. Every library,
, every environmental action group, every ar-
:very first-year student should have a copy."

Tony Ward, *Architectural Design*

he Oregon Experiment, shows how this theory
:mented, describing a new planning process for
y of Oregon.

Experiment is perhaps this decade's best can-
ermanently important book."

Robert Campbell, *Harvard Magazine*

) UNIVERSITY PRESS, NEW YORK

90000
9 780195 019193
ISBN 0-19-501919-9

A Pattern Language

Alexander Ishikawa Silverstein Jacobson Fiksdahl-King Angel

**

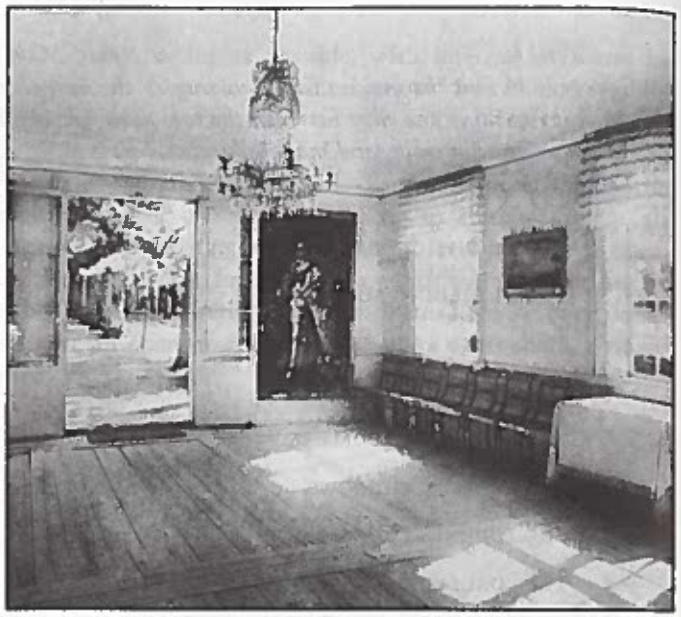
Oxford

A Pattern Language

Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

159 LIGHT ON TWO SIDES OF EVERY ROOM**



. . . once the building's major rooms are in position, we have to fix its actual shape: and this we do essentially with the position of the edge. The edge has got its rough position already from the overall form of the building—WINGS OF LIGHT (107), POSITIVE OUTDOOR SPACE (106), LONG THIN HOUSE (109), CASCADE OF ROOFS (116). This pattern now completes the work of WINGS OF LIGHT (107), by placing each individual room exactly where it needs to be to get the light. It forms the exact line of the building edge, according to the position of these individual rooms. The next pattern starts to shape the edge.

❖ ❖ ❖

When they have a choice, people will always gravitate to those rooms which have light on two sides, and leave the rooms which are lit only from one side unused and empty.

This pattern, perhaps more than any other single pattern, determines the success or failure of a room. The arrangement of daylight in a room, and the presence of windows on two sides, is fundamental. If you build a room with light on one side only, you can be almost certain that you are wasting your money. People will stay out of that room if they can possibly avoid it. Of course, if all the rooms are lit from one side only, people will have to use them. But we can be fairly sure that they are subtly uncomfortable there, always wishing they weren't there, wanting to leave—just because we are so sure of what people do when they do have the choice.

Our experiments on this matter have been rather informal and drawn out over several years. We have been aware of the idea for some time—as have many builders. (We have even heard that "light on two sides" was a tenet of the old Beaux Arts design tradition.) In any case, our experiments were simple: over and over again, in one building after another, wherever we happened to find ourselves, we would check to see if the pattern held. Were people in fact avoiding rooms lit only on one side, preferring the two-sided rooms—what did they think about it?

We have gone through this with our friends, in offices, in many homes—and overwhelmingly the two-sided pattern seems significant. People are aware, or half-aware of the pattern—they understand exactly what we mean.

With light on two sides . . . . and without

If this evidence seems too haphazard, please try these observations yourself. Bear the pattern in mind, and examine all the buildings you come across in your daily life. We believe that you will find, as we have done, that those rooms you intuitively recognize as pleasant, friendly rooms have the pattern; and those you intuitively reject as unfriendly, unpleasant, are the ones which do not have the pattern. In short, this one pattern alone, is able to distinguish good rooms from unpleasant ones.

The importance of this pattern lies partly in the social atmosphere it creates in the room. Rooms lit on two sides, with natural light, create less glare around people and objects; this lets us see things more intricately; and most important, it allows us to read in detail the minute expressions that flash across people's faces, the motion of their hands . . . and thereby understand, more clearly, the meaning they are after. *The light on two sides allows people to understand each other.*

In a room lit on only one side, the light gradient on the walls and floors inside the room is very steep, so that the part furthest from the window is uncomfortably dark, compared with the part near the window. Even worse, since there is little reflected light on the room's inner surfaces, the interior wall immediately next to the window is usually dark, creating discomfort and glare against this light. *In rooms lit on one side, the glare which surrounds people's faces prevents people from understanding one another.*

Although this glare may be somewhat reduced by supplementary artificial lighting, and by well-designed window reveals, the most simple and most basic way of overcoming glare, is to give every room two windows. The light from each window illuminates the wall surfaces just inside the other window, thus reducing the contrast between those walls and the sky outside. For details and illustrations, see R. G. Hopkinson, *Architectural Physics: Lighting*, London: Building Research Station, 1963, pp. 29, 103.

A supreme example of the complete neglect of this pattern is Le Corbusier's Marseilles Block apartments. Each apartment unit is very long and relatively narrow, and gets all its light from one end, the narrow end. The rooms are very bright just at the windows and dark everywhere else. And, as a result, the glare created by the light-dark contrast around the windows is very disturbing.

In a small building, it is easy to give every room light on two sides: one room in each of the four corners of a house does it automatically.

In a slightly larger building, it is necessary to wrinkle the edge, turn corners, to get the same effect. Juxtaposition of large rooms and small, helps also.

Wrinkle the edge.

In an even larger building, it may be necessary to build in some sort of systematic widening in the plan or to convolute the edge still further, to get light on two sides for every room.

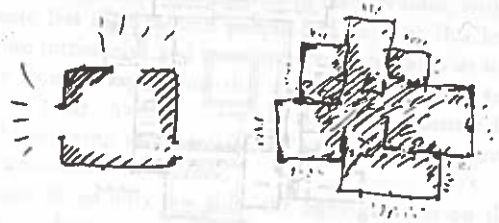
But of course, no matter how clever we are with the plan, no matter how carefully we convolute the building edge, sometimes it is just impossible. In these cases, the rooms can get the effect of light on two sides under two conditions. They can get it, if the room is very shallow—not more than about eight feet deep—with at least two windows side by side. The light bounces off the back wall, and bounces sideways between the two windows, so that the light still has the glare-free character of light on two sides.

And finally, if a room simply has to be more than eight feet deep, but cannot have light from two sides—then the problem can be solved by making the ceiling very high, by painting the walls very white, and by putting great high windows in the wall, set into very deep reveals, deep enough to offset the glare. Elizabethan dining halls and living rooms in Georgian mansions were often built like this. Remember, though, that it is very hard to make it work.

Therefore:

Locate each room so that it has outdoor space outside it on at least two sides, and then place windows in these outdoor walls so that natural light falls into every room from more than one direction.

each room has light on two sides



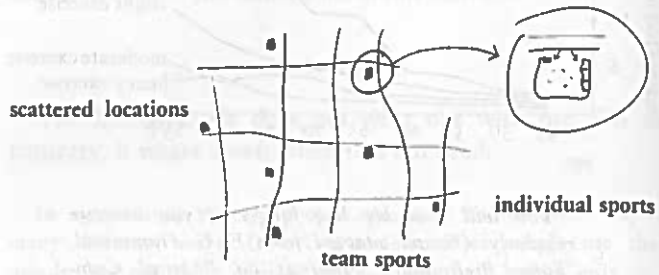
❖ ❖ ❖

Don't let this pattern make your plans too wild—otherwise you will destroy the simplicity of POSITIVE OUTDOOR SPACE (106), and you will have a terrible time roofing the building—ROOF

LAYOUT (209). Remember that it is possible to keep the essence of the pattern with windows on one side, if the room is unusually high, if it is shallow compared with the length of the window wall, the windows large, the walls of the room white, and massive deep reveals on the windows to make quite certain that the big windows, bright against the sky, do not create glare.

Place the individual windows to look onto something beautiful—WINDOWS OVERLOOKING LIFE (192), NATURAL DOORS AND WINDOWS (221); and make one of the windows in the room a special one, so that a place gathers itself around it—WINDOW PLACE (180). Use DEEP REVEALS (223) and FILTERED LIGHT (238). . . .

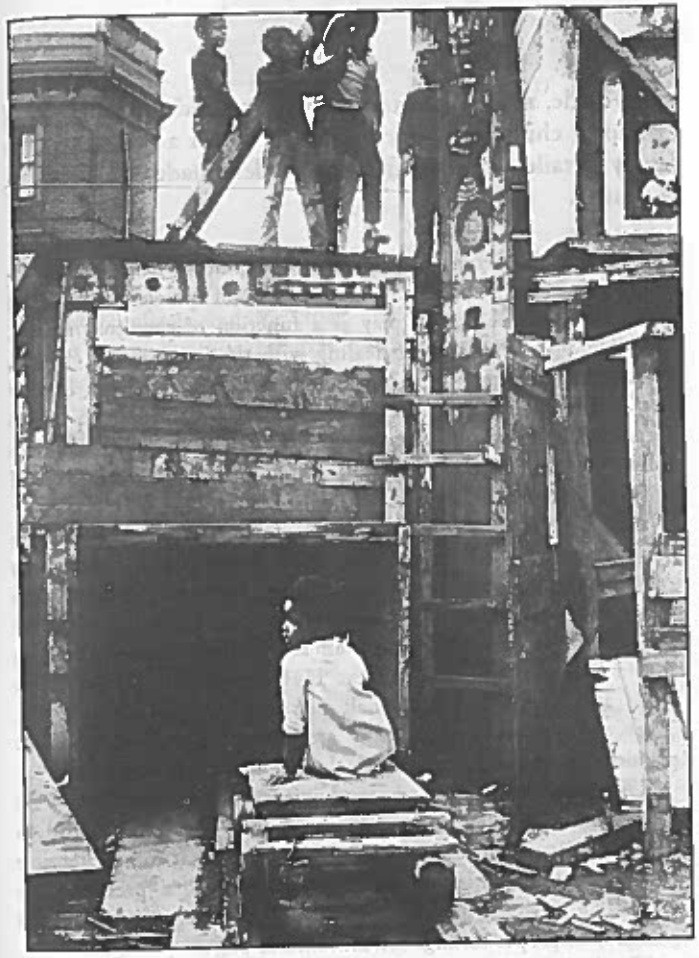
table tennis, swimming, billiards, basketball, dancing, gymnasium . . . and make the action visible to passers-by, as an invitation to participate.



❖ ❖ ❖

Treat the sports places as a special class of recognizable simple buildings, which are open, easy to get into, with changing rooms and showers—BUILDING COMPLEX (95), BATHING ROOM (144); combine them with community swimming pools, where they exist —STILL WATER (71); keep them open to people passing—BUILDING THOROUGHFARE (101), OPENING TO THE STREET (165), and provide places where people can stop and watch—SEAT SPOTS (241), SITTING WALL (243). . . .

73 ADVENTURE PLAYGROUND



. . . inside the local neighborhood, even if there is common land where children can meet and play—COMMON LAND (67), CONNECTED PLAY (68); it is essential that there be at least one smaller part, which is differentiated, where the play is wilder, and where the children have access to all kinds of junk.

✣ ✣ ✣

A castle, made of cartons, rocks, and old branches, by a group of children for themselves, is worth a thousand perfectly detailed, exactly finished castles, made for them in a factory.

Play has many functions: it gives children a chance to be together, a chance to use their bodies, to build muscles, and to test new skills. But above all, play is a function of the imagination. A child's play is his way of dealing with the issues of his growth, of relieving tensions and exploring the future. It reflects directly the problems and joys of his social reality. Children come to terms with the world, wrestle with their pictures of it, and reform these pictures constantly, through those adventures of imagination we call play.

Any kind of playground which disturbs, or reduces, the role of imagination and makes the child more passive, more the recipient of someone else's imagination, may look nice, may be clean, may be safe, may be healthy—but it just cannot satisfy the fundamental need which play is all about. And, to put it bluntly, it is a waste of time and money. Huge abstract sculptured playlands are just as bad as asphalt playgrounds and jungle gyms. They are not just sterile; they are useless. The functions they perform have nothing to do with the child's most basic needs.

This need for adventurous and imaginative play is taken care of handily in small towns and in the countryside, where children have access to raw materials, space, and a somewhat comprehensible environment. In cities, however, it has become a pressing concern. The world of private toys and asphalt playgrounds does not provide the proper settings for this kind of play.

The basic work on this problem has come from Lady Allen of

No playing.

Hurtwood. In a series of projects and publications over the past twenty years, Lady Allen has developed the concept of the adventure playground for cities, and we refer the reader, above all, to her work. (See, for example, her book, *Planning for Play*, Cambridge: MIT Press, 1968.) We believe that her work is so substantial, that, by itself, it establishes the essential pattern for neighborhood playgrounds.

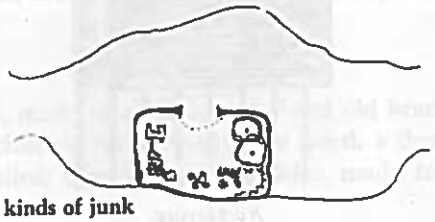
Colin Ward has also written an excellent review, "Adventure Playgrounds: A Parable of Anarchy," *Anarchy* 7, September 1961. Here is a description of the Grimsby playground, from that review:

> At the end of each summer the children saw up their shacks and shanties into firewood which they deliver in fantastic quantities to old age pensioners. When they begin building in the spring, "it's just a hole in the ground—and they crawl into it." Gradually the holes give way to two-storey huts. Similarly with the notices above their dens. It begins with nailing up "Keep Out" signs. After this come more personal names like "Bughold Cave" and "Dead Man's Cave," but by the end of the summer they have communal names like "Hospital" or "Estate Agent." There is an everchanging range of activities due entirely to the imagination and enterprise of the children themselves. . . .

Therefore:

Set up a playground for the children in each neighborhood. Not a highly finished playground, with asphalt and

swings, but a place with raw materials of all kinds—nets, boxes, barrels, trees, ropes, simple tools, frames, grass, and water—where children can create and re-create playgrounds of their own.

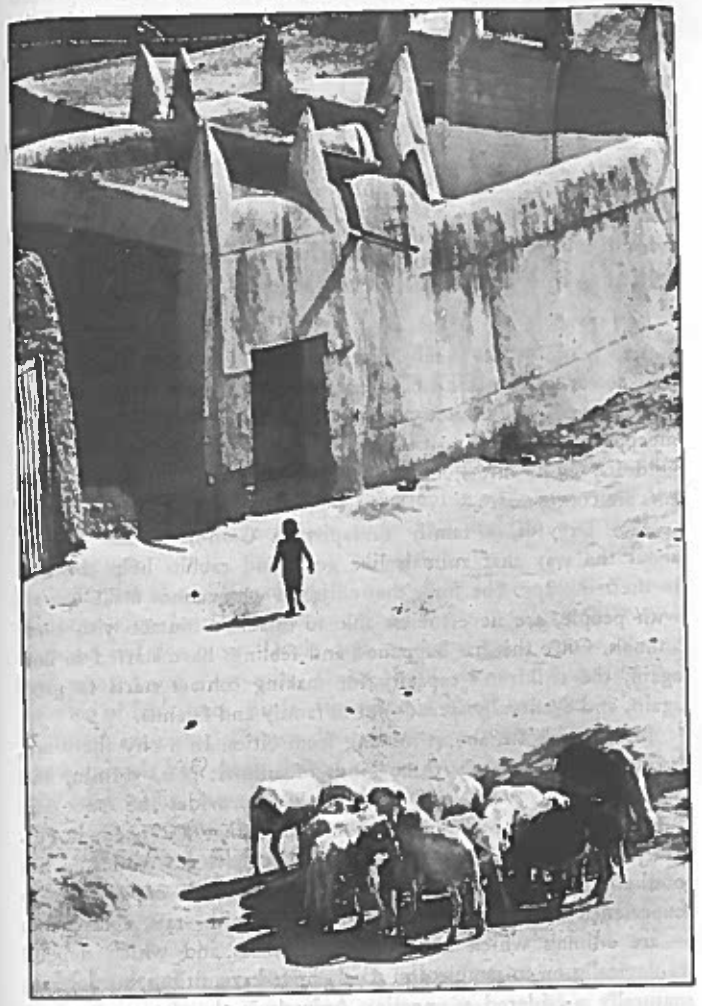


all kinds of junk

❖ ❖ ❖

Make sure that the adventure playground is in the sun—SUNNY PLACE (161); make hard surfaces for bikes and carts and toy trucks and trolleys, and soft surfaces for mud and building things—BIKE PATHS AND RACKS (56), GARDEN GROWING WILD (172), CHILD CAVES (203); and make the boundary substantial with a GARDEN WALL (173) or SITTING WALL (243). . . .

74 ANIMALS

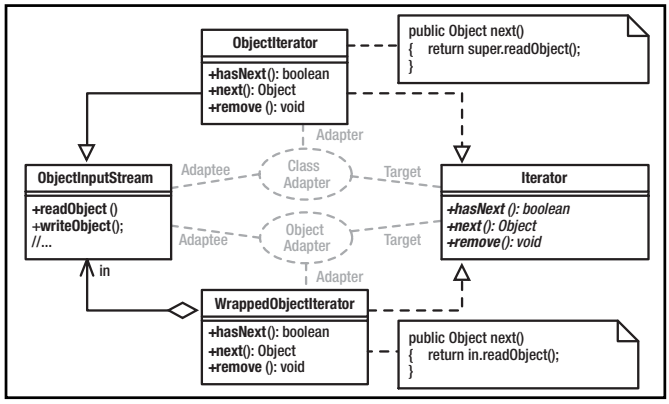


Structural Patterns

The structural patterns concern themselves with the organization of the program. I think of them as static-model patterns. Their intent is always to organize classes so that certain structural ends can be achieved. For example, the purpose of Bridge is to organize two subsystems in such a way that one subsystem can change radically (even be replaced entirely) without affecting the code in the other subsystem. The whole point of this organization is that you can make changes to the program without having to change the dynamic model at all.

Adapter

Make a class appear to support a familiar interface that it doesn't actually support. This way, existing code can leverage new, unfamiliar classes as if they are existing, familiar classes, eliminating the need to refactor the existing code to accommodate the new classes.



Adaptee: An object that doesn't support the desired interface

Target: The interface you want the Adaptee to support.

Adapter: The class that makes the Adaptee appear to support the Target interface. Class Adapters use derivation. Object Adapters use containment.

What Problem Does It Solve?

1. A library that you're using just isn't working out, and you need either to rewrite it or to buy a replacement from a third party and slot this replacement into your existing code, making as few changes as possible.
2. You may need to refactor a class to have a different interface than the original version (you need to add arguments to a method or change an argument or return-value type). You could have both old-style and new-style versions of the methods in one giant class, but it's better to have a single, simpler class (the new one) and use Adapter to make the new object appear to be one of the old ones to existing code.
3. Use an Adapter to make an old-style object serialized to disk appear to be a new-style object when loaded.

Pros (✔) and Cons (✖)

✔ Makes it easy to add classes without changing code.

✖ Identical looking Object and Class Adapters behave in different ways. For example, new `ObjectAdapter(obj)` and new `ClassAdapter(obj)` are both supported; the Object Adapter simply wraps `obj`, but the Class Adapter copies the fields of `obj` into its superclass component. Copying is expensive. On the plus side, a Class Adapter *is* an Adaptee, so it can be passed to methods expecting an object of the Adaptee class and also to methods that expect the Target interface. It's difficult to decide whether an Object or Class Adapter is best. It's a maintenance problem to have both.

✖ Difficult to implement when the library is designed poorly. For example, `java.io.InputStream` is an abstract class, not an interface, so you can't use the Class-Adapter pattern to create a `RandomAccessFile` that also supports the `InputStream` interface (you can't `extend` both `RandomAccessFile` and `InputStream`). You *can* use Object Adapter, or you can refactor the code to make `InputStream` an interface (as it should have been) and then implement that interface in an `AbstractInputStream` that has all the functionality now in `InputStream`. Collections do it correctly.

Often Confused With

Mediator: Mediator is the dynamic-model equivalent of Adaptor. Adapters are passive, passing messages to single Adaptees. Mediators interact with many colleagues in complex ways.

Bridge: Adapters change interfaces. Bridges isolate subsystems. Adapters are little things; Bridges are big.

Decorator: The encapsulated object in Decorator has the same interface as the container. Decorator modifies the behavior of some method or adds methods, but otherwise looks exactly like the wrapped object. Object Adapters have different interfaces than the wrapped object and don't change its behavior.

See Also

Mediator, Bridge, Decorator

Implementation Notes and Example

```
class ObjectIterator extends ObjectInputStream
                  implements Iterator
{  private boolean atEndOfFile = false;
   public ObjectIterator(InputStream src)
                       throws IOException
   {   super(src);
   }
   public boolean hasNext()
   {   return atEndOfFile == false;
   }
   public Object next()
   {   try
       {   return readObject();
       }
       catch( Exception e )
       {   atEndOfFile = true;
           return null;
       }
   }
   public void remove()
   {   throw new UnsupportedOperationException();
   }
}
class WrappedObjectIterator  implements Iterator
{  private boolean atEndOfFile = false;
   private final ObjectInputStream in;
   public
   WrappedObjectIterator(ObjectInputStream in)
   {   this.in = in;
   }
   public boolean hasNext()
   {   return atEndOfFile == false;
   }
   public Object next()
   {   try
       {   return in.readObject();
       }
       catch(Exception e){/* as above */}
   }
   public void remove()
   {   throw new UnsupportedOperationException();
   }
}
```

`ObjectIterator` is a Class Adapter that adapts an `ObjectInputStream` to implement the `Iterator` interface. This way, you can use existing methods that examine a set of objects by using an `Iterator` to examine objects directly from a file. The client doesn't know or care whether it's reading from a file or traversing a Collection of some sort. This flexibility can be useful when you're implementing an Object cache that can overflow to disk, for example. More to the point, you don't need to write two versions of the object-reader method, one for files and one for collections.

`WrappedObjectIterator` is an Object Adapter version of `ObjectIterator` that uses containment rather than inheritance.

The Class Adapter, since it *is* an `ObjectInputStream` that implements `Iterator`, can be used by any method that knows how to use either `ObjectInputStream` or Iterator. The Object Adapter, since it encapsulates the input stream, cannot be used as an `ObjectInputStream`, but you can use the input stream for a while, temporarily wrap it in a `WrappedObjectIterator` to extract a few objects, and then pull the input stream out again.

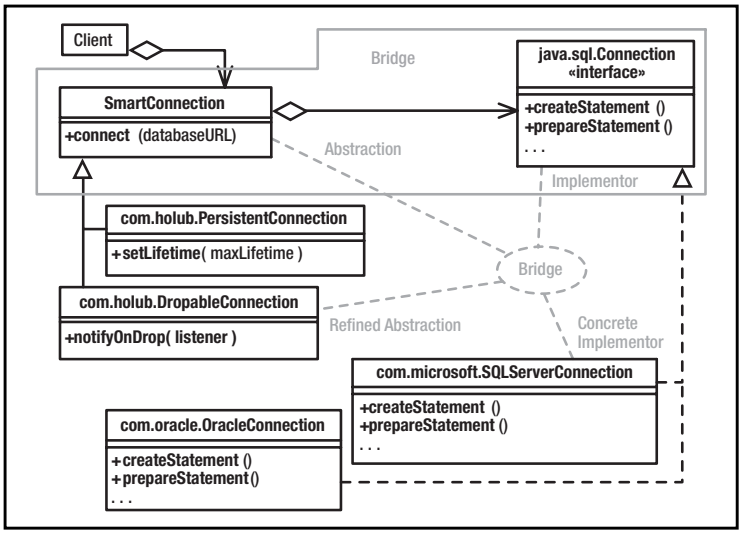
The two implementations require about the same amount of work so it's a judgment call which one is best. It all depends on what you're using it to do.

Usage

`InputStream in = new StringInputStream("hello");`	Adapter lets you access a `String` as if it were a file (`InputStream`). Similar adapters include `ByteArrayInputStream`, `CharArrayReader`, `PipedInputStream`, `PipedReader`, and `StringReader`. Don't confuse these adapters with the Decorators in *java.io* (`BufferedInputStream`, `PushbackInputStream`, and so on).

Bridge

To decouple subsystems so that either subsystem can change radically without impacting any code in the other one, put a set of interfaces between two subsystems and code to these interfaces.



Abstraction: A subsystem-independent portal into subsystem-specific code.

Implementor: An interface used by the Abstraction to talk to a subsystem-specific implementation. Typically is also the Abstract Product of an Abstract Factory.

Refined Abstraction: Often omitted, a version of the Abstraction, customized for a particular application.

Concrete Implementor: A subsystem-specific implementation of the Implementor.

What Problem Does It Solve?

Often used to achieve platform independence. Application-specific code on one side of the bridge uses platform-dependant code on the other side. Reimplement that interface, and the "business" logic doesn't know or care. Change the business logic, and the platform-specific interface implementations don't care. Often, you'll combine Bridge and Abstract Factory so that the Factory can supply the correct set of implementers at runtime, further isolating the two sides of the bridge. Examples of Bridge in Java are AWT and JDBC.

Pros (✔) and Cons (✖)

✔ In a pure inheritance model, you'd have a superclass that implemented some behavior and subclasses that customized this behavior for a specific platform. In Bridge, the superclass is effectively replaced by an interface, so the problems associated with implementation inheritance are minimized, and the total number of classes are reduced.

✖ It's difficult to implement interfaces so that each implementation behaves identically. Java's AWT Bridge implements windowing components for different operating environments, but the Motif implementation behaved differently on the screen than the Windows implementation.

Often Confused With

Bridge is more of an architecture than a design pattern. A Bridge is often a *set* of interfaces and classes (called *abstractions*, unfortunately—they're typically not `abstract`) that contain references to objects that implement a platform-independent interface in a platform-dependant way (Adapters). The Adapters are typically created by the Abstraction object using a Singleton-based Abstract Factory.

Adapter: Bridges separate subsystems, and Adapters make objects implement foreign interfaces. A one-interface bridge looks like a Class Adapter, however.

Facade: Facade simplifies the interface to a subsystem but may not isolate you from the details of how that subsystem works. Changes made on one side of the facade might mandate changes both to the other side of the facade and to the facade itself.

See Also

Abstract Factory, Singleton, Adapter, Facade, Mediator

Implementation Notes and Example

```
class SmartConnection
{   String username, password;
    java.sql.Connection connection;
    //...
    public void connect(String databaseURL)
                                  throws Exception
    {   Class.forName( databaseURL ).newInstance();

       Connection connection = null;
       Statement  statement  = null;
       //...
       connection =
             DriverManager.getConnection(
                 databaseURL, username, password );
    }
}

class PersistentConnection extends SmartConnection
{   long maxLifetime;
    public void setLifetime(long maxLifetime)
    {   // Arrange for connection to time
        // out after lifetime expires.
    }
}

class PooledConnection extends SmartConnection
{   public void notifyOnDrop(Runnable dropped)
      {   // Arange to call dropped.run()
          // when connection is dropped.
      }
}

//-----------------------------------------

class SQLServerConnection
               implements java.sql.Connection
{   // Implementation that support SQL Server
    // interface.
}

class OracleConnection implements
java.sql.Connection
{   // Implemenation that supports Oracle's
interface.
}
```

The abstraction classes (`SmartConnection`, `PersistentConnection`, and `DropableConnection`) use the Bridge interface (*java.sql.Connection*) to talk to the implementation classes (`OracleConnection`, `SQLServerConnection`).

The two sides of the Bridge can change independently. For example, I can change `OracleConnection` radically, and the classes on the other side of the Bridge (`SmartConnection`, for example) are completely unaware of that change. This isolation is possible because Factory is used to create the Concrete Implementers.

I can even support additional databases (by extending `java.sql.Connection`) without affecting the other side of the Bridge. By the same token, I can modify the `SmartConnection` class (and its subclasses) and even add additional subclasses, without impacting the other side of the bridge (the `java.sql.Connection` implementers).

Note that the Bridge completely isolates the subsystems from each other. The `Client` class knows only about the abstraction classes.

A Bridge is often very large. The JDBC Bridge consists of many Implementor interfaces and associated Concrete Implementations, and some of these interfaces are very large.

Chapter 1. Introduction

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?

One thing expert designers know *not* to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

An analogy will help illustrate the point. Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns like "represent states with objects" and "decorate objects so you can easily add/remove features." Once you know the pattern, a lot of design decisions follow automatically.

We all know the value of design experience. How many times have you had design *déjàvu*—that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it. However, we don't do a good job of recording experience in software design for others to use.

The purpose of this book is to record experience in designing object-oriented software as **design patterns**. Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively. To this end we have documented some of the most important design patterns and present them as a catalog.

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

None of the design patterns in this book describes new or unproven designs. We have included only designs that have been applied more than once in different systems. Most of these designs have never been documented before. They are either part of the folklore of the object-oriented community or are elements of some successful object-oriented systems—neither of which is easy for novice designers to learn from. So although these designs aren't new, we capture them in a new and accessible way: as a catalog of design patterns having a consistent format.

Despite the book's size, the design patterns in it capture only a fraction of what an expert might know. It doesn't have any patterns dealing with concurrency or distributed programming or real-time programming. It doesn't have any application domain-specific patterns. It doesn't tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database. Each of these areas has its own patterns, and it would be worthwhile for someone to catalog those too.

1.1 What Is a Design Pattern?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77, page *x*]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

In general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing

our catalog.

2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

   The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block. For this book we have concentrated on patterns at a certain level of abstraction. *Design patterns* are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*.

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample C++ and (sometimes) Smalltalk code to illustrate an implementation.

Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than procedural languages (Pascal, C, Ada) or more dynamic object-oriented languages (CLOS, Dylan, Self). We chose Smalltalk and C++ for pragmatic reasons: Our day-to-day experience has been in these languages, and they are increasingly popular.

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor (page 331). In fact, there are enough differences between Smalltalk and C++ to mean that some patterns can be expressed more easily in one language than the other. (See Iterator (257) for an example.)

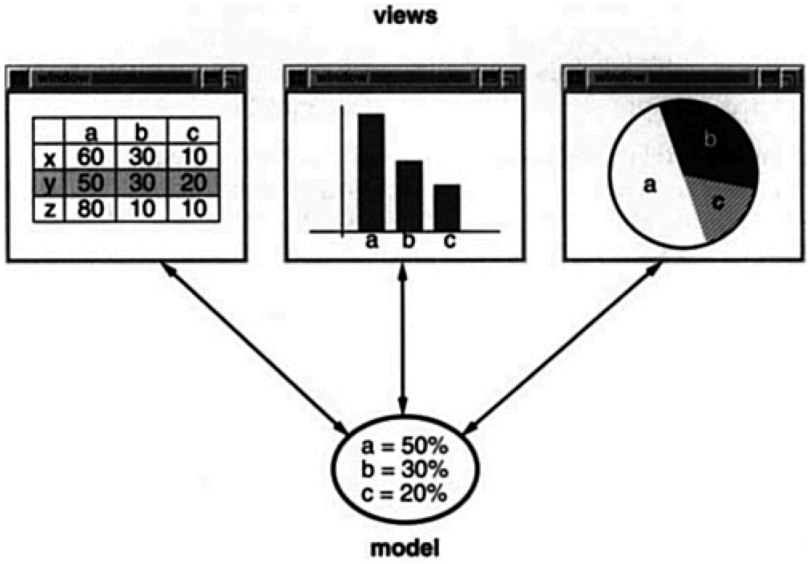
1.2 Design Patterns in Smalltalk MVC

The Model/View/Controller (MVC) triad of classes [KP88] is used to build user interfaces in Smalltalk-80. Looking at the design patterns inside MVC should help you see what we mean by the term "pattern."

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

The following diagram shows a model and three views. (We've left out the controllers for simplicity.) The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views communicate with the model to access these values.



Taken at face value, this example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the Observer (page 293) design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger. MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

Again, we could think of this as a design that lets us treat a composite view just like we treat one of its components. But the design is applicable to a more general problem, which occurs whenever we want to group objects and treat the group like an individual object. This more general design is described by the Composite (163) design pattern. It lets you create a class hierarchy in which some subclasses define primitive objects (e.g., Button) and other classes define composite objects (CompositeView) that assemble the primitives into more complex objects.

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way it responds to the keyboard, for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.

A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller. It's even possible to change a view's controller at run-time to let the view change the way it responds to user input. For example, a view can be disabled so that it doesn't accept input simply by giving it a controller that ignores input events.

The View-Controller relationship is an example of the Strategy (315) design pattern. A Strategy is an object that represents an algorithm. It's useful when you want to replace the algorithm either statically or dynamically, when you have a lot of variants of the algorithm, or when the algorithm has complex data structures that you want to encapsulate.

MVC uses other design patterns, such as Factory Method (107) to specify the default controller class for a view and Decorator (175) to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.

1.3 Describing Design Patterns

How do we describe design patterns? Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action.

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you 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 you recognize these situations?

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

Participants

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation

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

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns

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

The appendices provide background information that will help you understand the patterns and the discussions surrounding them. Appendix A is a glossary of terminology we use. We've already mentioned Appendix B, which presents the various notations. We'll also describe aspects of the notations as we introduce them in the upcoming discussions. Finally, Appendix C contains source code for the foundation classes we use in code samples.

1.4 The Catalog of Design Patterns

The catalog beginning on page 79 contains 23 design patterns. Their names and intents are listed next to give you an overview. The number in parentheses after each pattern name gives the page number for the pattern (a convention we follow throughout the book).

Abstract Factory (87) Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Adapter (139) Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge (151) Decouple an abstraction from its implementation so that the two can vary independently.

Builder (97) Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Chain of Responsibility (223) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command (233) Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Composite (163) Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator (175) Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade (185) Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Factory Method (107) Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Flyweight (195) Use sharing to support large numbers of fine-grained objects efficiently.

Interpreter (243) Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator (257) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator (273) Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento (283) Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer (293) Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Prototype (117) Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Proxy (207) Provide a surrogate or placeholder for another object to control access to it.

Singleton (127) Ensure a class only has one instance, and provide a global point of access to it.

State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

1.5 Organizing the Catalog

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. This section classifies design patterns so that we can refer to families of related patterns. The classification helps you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

We classify design patterns by two criteria (Table 1.1). The first criterion, called **purpose,** reflects what a pattern does. Patterns can have either **creational, structural**, or **behavioral** purpose. 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.

Table 1.1: Design pattern space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

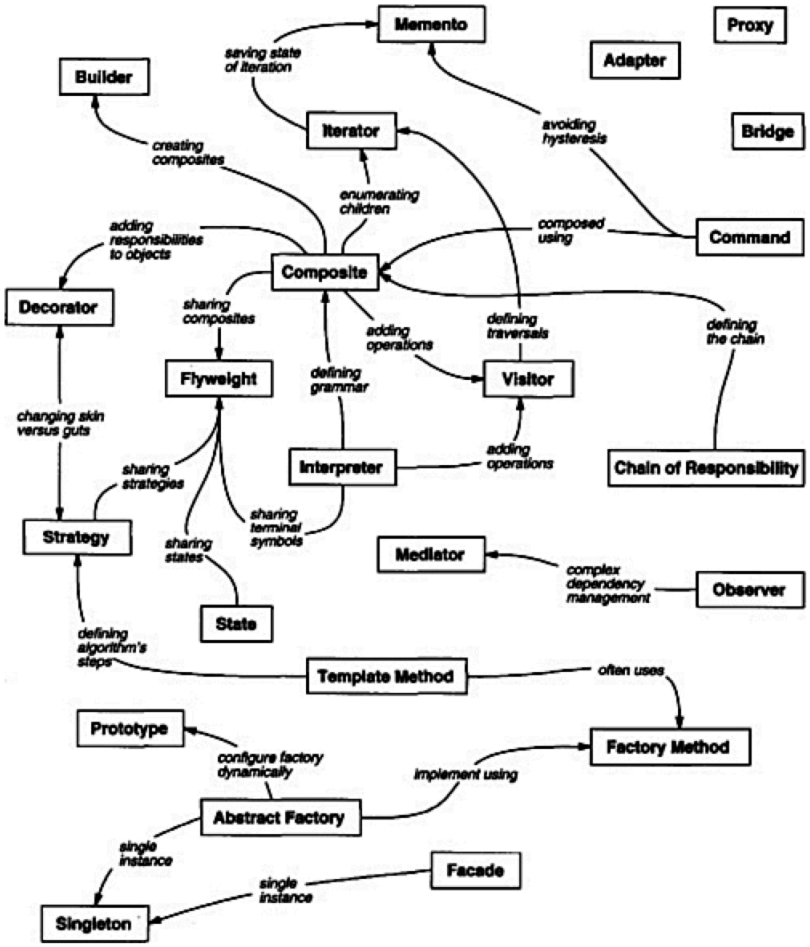
The second criterion, called **scope,** specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope.

Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

There are other ways to organize the patterns. Some patterns are often used together. For example, Composite is often used with Iterator or Visitor. Some patterns are alternatives: Prototype is often an alternative to Abstract Factory. Some patterns result in similar designs even though the patterns have different intents. For example, the structure diagrams of Composite and Decorator are similar.

Yet another way to organize design patterns is according to how they reference each other in their "Related Patterns" sections. Figure 1.1 depicts these relationships graphically.

Figure 1.1: Design pattern relationships



Clearly there are many ways to organize design patterns. Having multiple ways of thinking about patterns will deepen your insight into what they do, how they compare, and when to apply them.

1.6 How Design Patterns Solve Design Problems

Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. Here are several of these problems and how design patterns solve them.

Finding Appropriate Objects

Object-oriented programs are made up of objects. An **object** packages both data and the procedures that operate on that data. The procedures are typically called **methods** or **operations**. An object performs an operation when it receives a **request** (or **message)** from a **client**.

Requests are the *only* way to get an object to execute an operation. Operations are the *only* way to change an object's internal data. Because of these restrictions, the object's internal state is said to be **encapsulated**; it cannot be accessed directly, and its representation is invisible from outside the object.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on. They all influence the decomposition, often in conflicting ways.

Object-oriented design methodologies favor many different approaches. You can write a problem statement, single out the nouns and verbs, and create corresponding classes and operations. Or you can focus on the collaborations and responsibilities in your system. Or you can model the real world and translate the objects found during analysis into design. There will always be disagreement on which approach is best.

Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterparts in the real world. Some of these are low-level classes like arrays. Others are much higher-level. For example, the Composite (163) pattern

introduces an abstraction for treating objects uniformly that doesn't have a physical counterpart. Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.

Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy (315) pattern describes how to implement interchangeable families of algorithms. The State (305) pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

Determining Object Granularity

Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide what should be an object?

Design patterns address this issue as well. The Facade (185) pattern describes how to represent complete subsystems as objects, and the Flyweight (195) pattern describes how to support huge numbers of objects at the finest granularities. Other design patterns describe specific ways of decomposing an object into smaller objects. Abstract Factory (87) and Builder (97) yield objects whose only responsibilities are creating other objects. Visitor (331) and Command (233) yield objects whose only responsibilities are to implement a request on another object or group of objects.

Specifying Object Interfaces

Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's **signature**. The set of all signatures defined by an object's operations is called the **interface** to the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object.

A **type** is a name used to denote a particular interface. We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window." An object may have many types, and widely different objects can share a type. Part of an object's interface may be characterized by one type, and other parts by other types. Two objects of the same type need only share parts of their interfaces. Interfaces can contain other interfaces as subsets. We say that a type is a **subtype** of another if its interface contains the interface of its **supertype**. Often we speak of a subtype *inheriting* the interface of its supertype.

Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation —different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces.

When a request is sent to an object, the particular operation that's performed depends on *both* the request *and* the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request to an object and one of its operations is known as **dynamic binding**.

Dynamic binding means that issuing a request doesn't commit you to a particular implementation until run-time. Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request. Moreover, dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as **polymorphism**, and it's a key concept in object-oriented systems. It lets a client object make few assumptions about other objects beyond supporting a particular interface. Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.

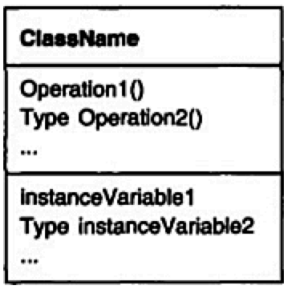
Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what *not* to put in the interface. The Memento (283) pattern is a good example. It describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later. The pattern stipulates that Memento objects must define two interfaces: a restricted one that lets clients hold and copy mementos, and a privileged one that only the original object can use to store and retrieve state in the memento.

Design patterns also specify relationships between interfaces. In particular, they often require some classes to have similar interfaces, or they place constraints on the interfaces of some classes. For example, both Decorator (175) and Proxy (207) require the interfaces of Decorator and Proxy objects to be identical to the decorated and proxied objects. In Visitor (331), the Visitor interface must reflect all classes of objects that visitors can visit.

Specifying Object Implementations

So far we've said little about how we actually define an object. An object's implementation is defined by its **class**. The class specifies the object's internal data and representation and defines the operations the object can perform.

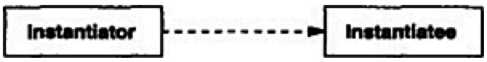
Our OMT-based notation (summarized in Appendix B) depicts a class as a rectangle with the class name in bold. Operations appear in normal type below the class name. Any data that the class defines comes after the operations. Lines separate the class name from the operations and the operations from the data:



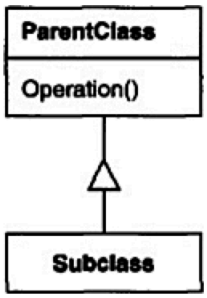
Return types and instance variable types are optional, since we don't assume a statically typed implementation language.

Objects are created by **instantiating** a class. The object is said to be an **instance** of the class. The process of instantiating a class allocates storage for the object's internal data (made up of **instance variables**) and associates the operations with these data. Many similar instances of an object can be created by instantiating a class.

A dashed arrowhead line indicates a class that instantiates objects of another class. The arrow points to the class of the instantiated objects.



New classes can be defined in terms of existing classes using **class inheritance**. When a **subclass** inherits from a **parent class**, it includes the definitions of all the data and operations that the parent class defines. Objects that are instances of the subclass will contain all data defined by the subclass and its parent classes, and they'll be able to perform all operations defined by this subclass and its parents. We indicate the subclass relationship with a vertical line and a triangle:



An **abstract class** is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated. The operations that an abstract class declares but doesn't implement are called **abstract operations**. Classes that aren't abstract are called **concrete classes**.

Subclasses can refine and redefine behaviors of their parent classes. More specifically, a class may **override** an operation defined by its parent class. Overriding gives subclasses a chance to handle requests instead of their parent classes. Class inheritance lets you define classes simply by extending other classes, making it easy to define families of objects having related functionality.