Chapter 15

Design Patterns

Like most complex structures, good computer programs are often formed by imitating the structure of older, similar programs that have already proven successful. The concept of a *design pattern* is an attempt to capture and formalize this process of imitation. The basic idea is to characterize the features of a proven solution to a small problem, summarizing the essential elements and omitting the unnecessary detail. A catalog of design patterns is a fascinating illustration of the myriad ways that software can be structured so as to address different problems. Later, patterns can give insight into how to approach new problems that are similar to those situations described by the pattern.

This chapter will introduce the idea of design patterns by describing several that are found in the Java library. The terminology used in describing the patterns is adapted from the book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [Gamma 1995]. This was one of the first books to describe the concept of design patterns and provide a systematic cataloging of patterns. Many more patterns than are described here can be found in this book, as well as in the recent literature on design patterns.

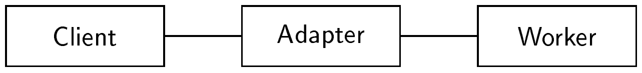
The format used in describing each pattern is to first characterize the problem the pattern is addressing. Then, the essential features of the solution are summarized. In some cases this is followed by a discussion that examines some of the context for the problem, or contrasts alternative design possibilities. This is followed by a more detailed description of the pattern as it is manifest in the Java Library. Finally, a sentence or two summarizes the situations where the pattern is applicable.

15.1 Adapter

problem: How do you use an object that provides appropriate behavior but uses a different interface than is required in some situation?

solution: Define an adapter class that acts as an intermediary. The adapter does little work itself, but merely translates commands from one form into the other.

discussion: International travelers frequently overcome the problem of differing electrical plug and voltage standards by using adapters for their appliances. These adapters allow an electrical appliance that uses one type of plug to be modified so that it can be used with a different type of plug. The software equivalent is similar. An adapter is concerned mostly with changes in the interface to an object, and less with the actual functionality being provided.



example: An example of adapters in the Java library are the "wrapper" classes, Boolean, Integer, and so on. These adapt a primitive type (such as boolean or int) so that they can be used in situations where an Object is required. For example, wrappers are necessary to store primitive values as elements in a Vector.

Another form of adapter is the class MouseAdapter used in the pin ball game described in Chapter 7, as well as in the Solitare program presented in Chapter 9. Here the adapter reduces the interface, by implementing default behavior for methods that are unneeded in the current application. The client can therefore concentrate on the one method that is used in the program.

An adapter can be used whenever there is the need for a change in interface, but no, or very little, additional behavior beyond that provided by the worker.

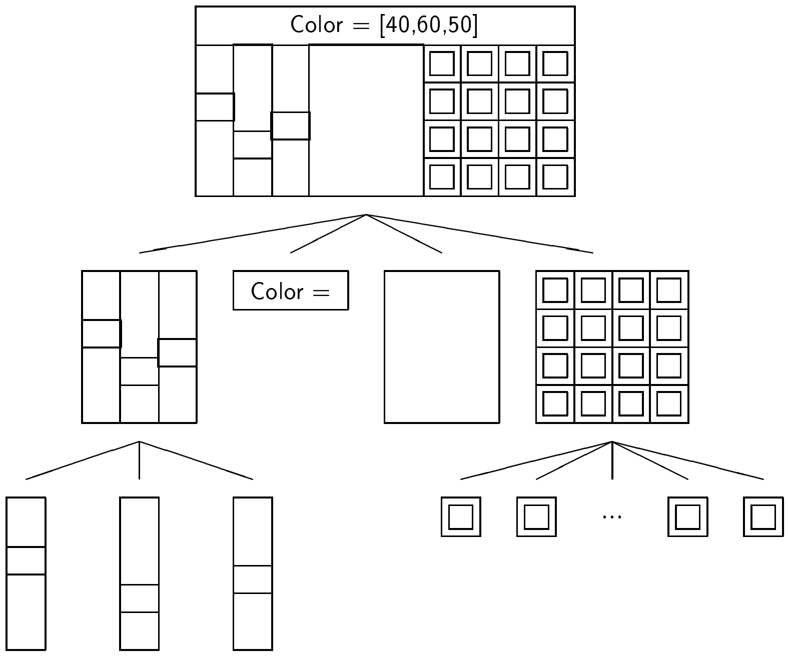
15.2 Composition

problem: How do you permit the creation of complex objects using only simple parts?

solution: Provide a small collection of simple components, but also allow these components to be nested arbitrarily. The resulting composite objects allow individual objects and compositions of objects to be treated uniformly. Frequently, an interesting feature of the composition pattern is the merging of the *is-a* relation with the *has-a* relation.

example: A good example of composition in the Java library is the creation of design layouts through the interaction of Components and Containers. There are only five simple types of layouts provided by the standard library, and of these five only two, border layouts and grid layouts, are commonly used. Each item in a layout is a Component. Composition occurs because Containers are also Components. A container holds its own layout, which is again one of only a few simple varieties. Yet the container is treated as a unit in the original layout.

The structure of a composite object is often described in a tree-like format. Consider, for example, the layout of the window shown in Figure 13.8 of Chapter 13. At the application level there are four elements to the layout. These are a text area, a simple blank panel, and two panels that hold composite objects. One of these composite panels holds three scroll bars, while the second is holding a grid of sixteen buttons.



By nesting panels one within another, arbitrarily complex layouts can be created.

Another example of composition is the class SequenceInputStream, which is used to catenate two or more input streams so that they appear to be a single input source (see Section 14.1.2). A SequenceInputStream *is-a* InputStream (meaning it extends the class InputStream). But a SequenceInputStream also *has-a* InputStream as part of its internal state. By combining inheritance and composition, the class permits multiple sequences of input sources to be treated as a single unit.

This pattern is useful whenever it is necessary to build complex structures out of a few simple elements. Note that the merging of the *is-a* and *has-a* relations is characteristic of the *wrapper* pattern (Section 15.9), although wrappers can be constructed that are not composites.

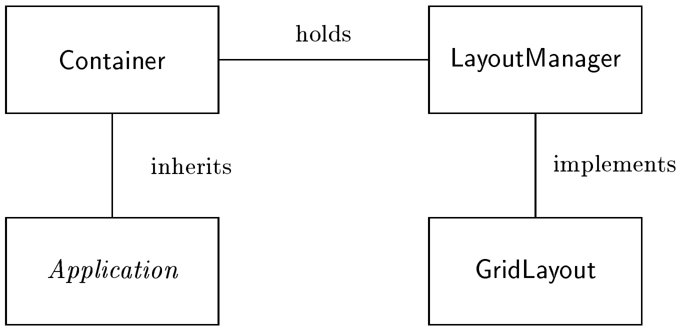
15.3 Strategy

problem: How do you allow the algorithm that is used to solve a particular problem to be easily and dynamically changed by the client?

solution: Define a family of algorithms with a similar interface. Encapsulate each algorithm, and let the client select the algorithm to be used in any situation.

discussion: If a complex algorithm is embedded in a larger application, it may be difficult to extract the algorithm and replace it with another, alternative version. If several alternative algorithms are included in the same object, both the complexity and the code of the resulting object may be increased unnecessarily. By separating problem and solution, it is easier for the client to select the solution (algorithm) appropriate for any particular situation.

example: An example of the use of the *Strategy* pattern is the creation of layout managers in the AWT. Rather than coding in the component library the details of how items are laid out on the screen, these decisions are left to the layout manager. An interface for LayoutManager is defined, and five standard layout managers are provided. The ambitious programmer is even allowed, should he or she choose, to define a new object that satisfies the LayoutManager interface.



The activities of the design component (such as a Panel or a Window) is independent of the particular layout manager that is being used. This both simplifies the container component and permits a much greater degree of flexibility in the structure of the resulting layout than would be possible if layout decisions were an intrinsic part of the container.

This pattern is useful whenever it is necessary to provide a set of alternative solutions to a problem, and the algorithms used to address the problem can be encapsulated with a simple interface.

15.4 Observer

problem: How do you allow two or more independent and loosely coupled objects to change in synchrony with each other?

solution: Maintain a list of objects that are tied, or dependent, on another object. When the target object changes, the dependents are notified that they should update themselves.

discussion: It is easy to maintain tightly coupled objects in synchrony. For example, if a new class is defined as a subclass of an existing parent class, modifications of the parent that are made via method invocations can be monitored by simply overriding the methods. It is much more difficult to keep objects in step with each other when links are formed and broken dynamically at run time, or when no obvious class relationship exists between the separate elements.

example: There are two good examples of the use of the observer pattern in the Java library. The first we have seen in many earlier case studies, such as the cannon world examined in Chapter 6. Each of the user interface components that permits interaction, such as buttons, scroll bars, and check boxes, maintains a collection of *listener* objects. This list is dynamic; listeners for any component can be easily added or removed at run time. Furthermore, the structure of the listeners is not specified, they are only required to satisfy the necessary interface. When the component changes state (the button is pressed, the slider moved, the Checkbox changed), each of the listeners is notified that a change has occurred. It is up to the listener to decide what action should be taken as a result of the change.

The idea behind listeners is also found in a more general facility that can be used by programmers for situations that do not involve user interaction. The library class Observable represents objects that can be "observed", the equivalent of the components in the AWT mechanism. Programmers can either subclass a new class from Observable, or simply create an Observable field within a class. Other objects can implement the Observer interface. These correspond to "listener" objects. An instance of Observer registers itself with the object being observed.

At any time, the Observable object can indicate that it has changed, by invoking the message notifyObservers(). An optional argument can be passed along with this message. Each observer is passed the message update(Observable, Object), where the first argument is the Observable that has changed, and the second is the optional argument provided by the notification. The observer takes whatever action is necessary to bring the state into synchrony with the observed object.

The *Observer* pattern is applicable whenever two or more objects must be loosely coupled, but must still maintain synchronization in some aspect of their behavior or state.

15.5 Flyweight

problem: How can one reduce the storage costs associated with a large number of objects that have similar state?

solution: Share state in common with similar objects, thereby reducing the storage required by any single object.

example: With the exception of primitive values, all objects in Java are an instance of some class. With each class it is necessary to associate certain information. Examples of information is the name of the class (a String), and the description of the interface for the class. If this information was duplicated in each object the memory costs would be prohibitive. Instead, this information is defined once by an object of type Class, and each instance of the class points to this object.

The objects that share the information are known as *flyweights*, since their memory requirements are reduced (often dramatically) by moving part of their state to the shared value. The flyweight pattern can be used whenever there are a large number of objects that share a significant common internal state.

15.6 Abstract Factory

problem: How to provide a mechanism for creating instances of families of related objects without specifying their concrete representations.

solution: Provide a method that returns a new value that is characterized only by an interface or parent class, not by the actual type produced.

discussion: There are several instances where the value returned by a function in the standard library is characterized by either an abstract class or an interface. Clearly the actual value being returned is a different type, but normally the client using the function is not concerned with the actual type, but only the behavior described by the characterizing attributes.

example: Two examples out of the many found in the Java library will be described. Each of the collection classes Vector, Hashtable and Dictionary define a method named elements() that is described as returning a value of type Enumeration. As Enumeration is only an interface, not a class, the value returned is clearly formed as an instance of some other class. Almost always, the client has no interest in the actual type being yielded by elements(), and is only interested in the behavior common to all values that satisfy the Enumeration interface.

A similar situation occurs with the classes Font and FontMetrics. The class FontMetrics is used to describe the characteristics of a Font, such as the height and width of characters, the distance characters extend above or below the baseline, and so on. A FontMetrics is an abstract class, one that cannot be instanciated directly by the programmer using the new command. Instead, a value of type FontMetric is returned by a Graphics object in response to the message getFontMetrics. Clearly, the graphics object is returning a value derived from a subclass of FontMetric, but the particular value returned is normally of no concern to the client.

A similar facility is used by class Applet, which can return an AppletContext that describes the current execution environment.

The abstract factory pattern should be used whenever the type of the actual value to be created cannot be predicted in advance, and therefore must be determined dynamically.

15.7 Factory Method

problem: You have a method that returns a newly created object, but want subclasses to have the ability to return different types of object.

solution: Allow the subclass to override the creation method and return a different type of object.

discussion: This pattern is very similar to the abstract factory, only specialized for the situation where new abstractions are formed using inheritance.

example: The method clone() is a good example of a factory method. This method returns a copy of an object, provided the object supports the Cloneable interface. The default method in class Object raises an exception, indicating that the cloneable interface is not supported. Subclasses that wish to permit clones must override this method, and return a different type of value.

Note that the value returned by a factory method must be the same for all classes. For the Cloneable interface this type is Object. Any class that permits cloning will still return a value of type Object in response to the message clone(). This value must then be cast to the appropriate type.

The factory method pattern is useful when there is a hierarchy of abstractions formed using inheritance, and part of the behavior of these abstractions is the creation of new objects.

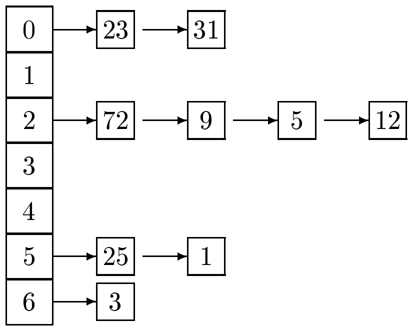
15.8 Iterator

problem: How to provide a way to access elements of an aggregate object sequentially without exposing the underlying representation.

solution: Provide a mediator object for the sole purpose of sequential access. This mediator can be aware of the representation of the aggregate, however the client using the object need not be aware of these details.

example: The Enumeration interface for container access actually addresses two related problems. It provides a uniform means of accessing elements from many different types of container, and it hides the details of the underlying container representation. It is the second aspect that makes the Enumeration a good example of the iterator pattern.

Consider, for example, an enumeration that is generating elements from a Hashtable. Internally, a hash table is implemented as an array, each element of the array being a list. Values that hash into the same locations are found on the same list.



The programmer who uses a hash table and wishes to iterate over the values should not be concerned with the representation, such as moving from one list to the next when the elements in one hash location have been exhausted. The hash table enumeration hides these difficulties behind a simple interface. The programmer sees only the two methods hasMoreElements() and nextElement(). With these, a loop can be written that does not even hint at the complex actions needed to access the underlying elements.

```
HashTable htab = new HashTable();
...
for (Enumeration e = htab.elements(); e.hasMoreElements(); ) {
    Object val = e.nextElement();
    ...
    }
```

The fact that the method elements returns a value that is not directly an Enumeration, but is rather a value from another class that implements the Enumeration interface, is an example of the *Abstract Factory* pattern (Section 15.6).

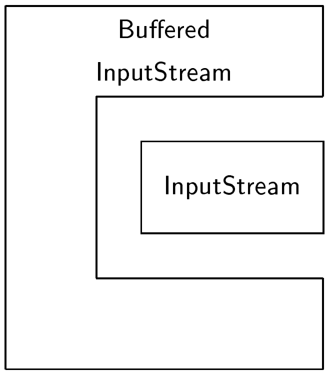
The iterator pattern is useful whenever an aggregate object is created that can hold an arbitrary number of values, and it is necessary to provide access to values without exposing the underlying representation.

15.9 Decorator (Filter or Wrapper)

problem: How can you attach additional responsibilities to an object dynamically?

solution: By combining the *is-a* and *has-a* relations, create an object that wraps around an existing value, adding new behavior without changing the interface.

discussion: Inheritance is one technique for providing new functionality to an existing abstraction. But inheritance is rather heavy handed, and is often not flexible enough to accommodate situations that must dynamically change during the course of execution. A decorator wraps around an existing object, and satisfies the same requirements (for example, is subclassed from the same parent class or implements the same interface). The wrapper delegates much of the responsibility to the original, but occasionally adds new functionality.



example: The class InputStream provides a way to read bytes from an input device, such as a file. The class BufferedInputStream is a subclass of InputStream, adding the ability to buffer the input so that it can be reset to an earlier point and values can be reread two or more times. Furthermore, a BufferedInputStream can take an InputStream as argument in its constructor.

Because a BufferedInputStream both *is* an InputStream and *has* an input stream as part of its data, it can be easily wrapped around an existing input stream. Due to inheritance and substitutability, the BufferedInputStream can be used where the original InputStream was expected. Because it holds the original input stream, any actions unrelated to the buffering activities are simply passed on to the original stream.

A decorator, or wrapper class, is often a flexible alternative to the use of subclassing. Functionality can be added or removed simply by adding or deleting wrappers around an object.

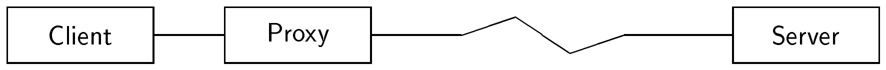
15.10 Proxy

problem: How do you hide details such as transmission protocols to remote objects?

solution: Provide a proxy that acts as a surrogate or placeholder for another object.

discussion: The idea of a proxy is that one object is standing in place of another. The first object receives requests for the second, and generally forwards the requests to the second, after processing them in some fashion.

example: An example proxy in the Java Library is the RMI, or Remote Method Invocation system. The RMI is a mechanism that can be used to coordinate Java programs running on two or more machines. Using the RMI, a proxy object is created that runs on the same machine as the client. When the client invokes a method on the proxy, the proxy transmits the method across the network to the server on another machine. The server handles the request, then transmits the result back to the proxy. The proxy hands the result back to the client. In this fashion, the details of transmission over the network are handled by the proxy and the server, and are hidden from the client.



15.11 Bridge

problem: How to decouple an abstraction from its implementation so that the latter can vary independently.

solution: Remove implementation details from the abstraction, placing them instead in an object that is held as a component in the abstraction.

example: Most of the component classes in the AWT make use of the bridge pattern. Fundamentally, this is because the actions necessary to implement a graphical component vary in great detail from one platform to another. For example, the actions needed to display a window are different depending upon whether the underlying display is X-Windows/Motif, Windows-95, or the Macintosh. Rather than placing platform specific details in the class Window, instead each window maintains a component of type WindowPeer. The interface WindowPeer has different implementations, depending upon the platform on which the application is being executed. This separation allows a Java program that depends only on the class Window to be executed in any environment for which there is a corresponding peer.

The Bridge pattern is in many ways similar to the Strategy pattern described earlier. Differences are that bridges are almost always hidden from the client (for example, the average Java programmer is generally unaware of the existence of the peer classes), and are generally dictated by environmental issues rather than reflecting design decisions.

15.12 Chapter Summary

An emerging new area of study in object-oriented languages is the concept of design patterns. A design pattern captures the salient characteristics of a solution to a commonly observed problem, hiding details that are particular to any one situation. By examining design patterns, programmers learn about techniques that have proven to be useful in previous problems, and are therefore likely to be useful in new situations.

Further Reading

The most important reference for design patterns is the book of the same name [Gamma 1995], by Gamma, Helm, Johnson and Vlissides (commonly known as the Gang of Four, or GOF). Another recent book on patterns is by RichardndexGabriel, Richard Gabriel [Gabriel 1996].

Study Questions

1. In what ways is an adapter similar to a proxy? In what ways are they different?
2. In what way is the composition design pattern similar to the idea of composition examined in Chapter 10?
3. In what ways is a strategy similar to a bridge? In what ways are they different?
4. In what ways is an iterator similar to an adapter?

Exercises

1. What design pattern is exhibited by the class PrintStream (see Section 14.2)? Explain your answer.