C H A P T E R 4

Fundamental Design Patterns

The patterns in this chapter are the most fundamental and important design patterns to know. You will find these patterns used extensively in other design patterns.

The Delegation; Interface; Abstract Superclass; and Interface and Abstract Class patterns demonstrate how to organize relationships between classes. Most patterns use at least one of these patterns. They are so ubiquitous that they are often not mentioned in the Related Patterns sections for most patterns.

The Immutable pattern describes a way to avoid bugs and delays when multiple objects access the same object. Although the Immutable

pattern is not explicitly part of the majority of other patterns, it can be used advantageously with most patterns.

The Marker Interface pattern describes a way to simplify the design of classes that have a constant boolean attribute.

The Proxy pattern is the basis for a number of patterns that share the common concept in which an object manages access to another object in a relatively transparent way.

Delegation (When Not to Use Inheritance)

SYNOPSIS

In some situations, using inheritance to extend a class leads to a bad design. Though less convenient, delegation is a more general-purpose way of extending classes. Delegation succeeds in many situations where inheritance does not work well.

CONTEXT

Inheritance is a common way to extend and reuse the functionality of a class. Delegation is a more general way for extending a class's behavior that involves a class calling another class's methods rather than inheriting them. Inheritance is inappropriate for many situations in which delegation is appropriate.

For example, inheritance is useful for capturing "is-a-kind-of" relationships because of their static nature. However, "is-a-role-played-by" relationships are awkward to model by inheritance. Instances of a class can play multiple roles. Consider the example of an airline reservation system that includes such roles as passenger, ticket-selling agent, and flight crew. It's possible to represent these role relationships as a class called `Person` that has subclasses corresponding to these roles, as shown in Figure 4.1.

The problem with the model in Figure 4.1 is that the same person can fill more than one of these roles. A person who is normally part of a flight crew can also be a passenger. Some airlines occasionally float flight crew members to the ticket counter, which means that the same person can fill

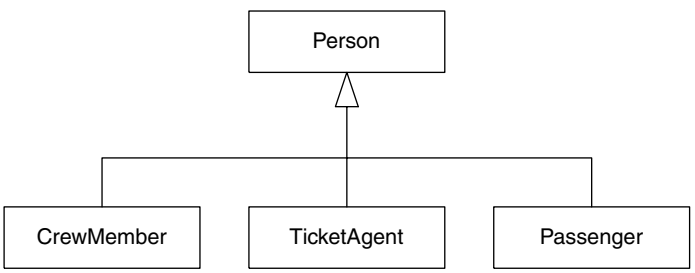

FIGURE 4.1 Modeling roles with inheritance.

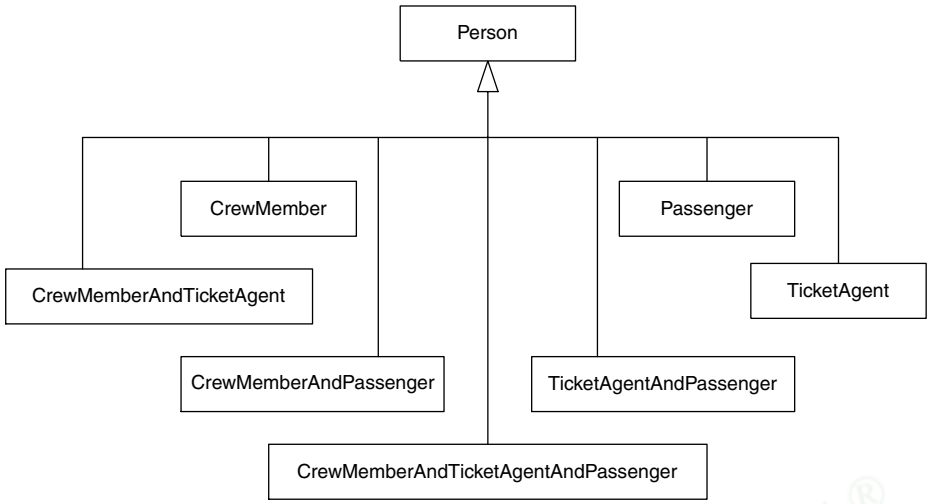

FIGURE 4.2 Modeling multiple roles with Inheritance.

any combination of these roles. To model this situation, you would need seven subclasses for `Person`, as shown in Figure 4.2. The number of subclasses needed increases exponentially with the number of roles. To model all the combinations of six roles would require 63 subclasses.

A more serious problem is that the same person may play different combinations of roles at different times. Inheritance relationships are static and do not change over time. To model different combinations of roles over time using inheritance relationships, it is necessary to use different objects at different times to represent the same person in order to capture changes in role. Modeling dynamically changing roles with inheritance gets complicated.

On the other hand, it is possible to represent persons in different roles using delegation without having any of these problems. Figure 4.3 shows how the model could be reorganized using delegation.

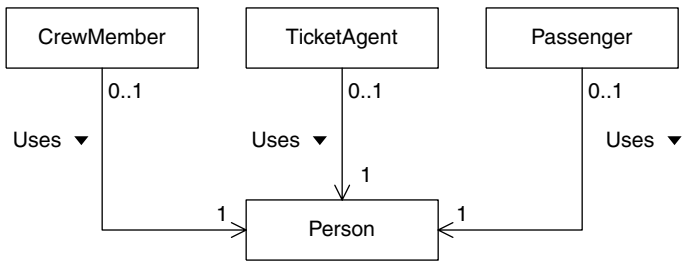

FIGURE 4.3 Modeling roles with delegation.

Using the organization shown in Figure 4.3, a `Person` object delegates the responsibility of filling a particular role to an object that is specific to that role. You need only as many objects as there are roles to fill. Different combinations do not require additional objects. Because delegation objects can be dynamic, role objects can be added or removed as a person fills different roles.

In the case of the airline reservation software, a predetermined set of role objects may become associated with different `Person` objects over time. For example, when a flight is scheduled, it will be determined that a certain number of flight crewmembers must be on board. When a flight is staffed, specific persons will be associated with crewmember roles. As schedules change, a person may be shifted from one crewmember role to another.

FORCES

- ☺ Inheritance is a static relationship; it does not change over time. If it is found that an object needs to be a different subclass of a class at different times, it should not have been made a subclass of that class in the first place. If an object is created as an instance of a class, it will always be an instance of that class. However, an object can delegate behavior to different objects at different times.
- ☺ If it is found that a class attempts to hide a method or variable inherited from a superclass from other classes, that class should not inherit from the superclass. There is no effective way to hide methods or variables inherited from a superclass. However, it is possible for an object to use another object's methods and variables while ensuring that it is the only object with access to the other object. This accomplishes the same thing as inheritance but uses dynamic relationships that can change over time.
- ☺ A class related to a program's problem domain should not be a subclass of a utility class. There are two reasons for this:
  - If a class is declared a subclass of a class such as `ArrayList` or `HashMap`, there is risk that these classes not under your control will change in an incompatible way. Though the risk is low, usually there is no corresponding benefit to offset it.
  - When a problem-domain-specific class as a subclass of a utility class is written, usually the intention is to use the functionality of the utility class for implementing problem-domain-specific functionality. The problem with using inheritance this way is that it weakens the encapsulation of the problem domain class's implementation.

Client classes that use the problem-domain-specific class may be written in a way that assumes the problem-domain-specific class to be a subclass of the utility class. If a change to the implementation of the problem domain class gives it a different superclass, client classes that rely on the problem-domain class having its original superclass will break.

An even more serious problem is that client classes can call the public methods of the utility superclass, thereby defeating its encapsulation.

- ☹ Delegation can be less convenient than inheritance, because it requires more code to implement.
- ☹ Delegation imposes less structure on classes than inheritance. In designs in which constraining the structure of classes is important, the structure and inflexibility of inheritance may be a virtue. This is often true in frameworks. See the discussion of the Template Method pattern for more details of this issue.

Some inappropriate uses of inheritance are sufficiently common to be classified as AntiPatterns. In particular, subclassing utility classes and using inheritance to model roles are common design flaws.

- ☺ Most reuse and extension of a class is not appropriately done through inheritance.
- ☺ The behavior that a class inherits from its superclass cannot be easily changed over time. Inheritance is not useful when the behavior on which a class should build is not determined until runtime.

SOLUTION

Use delegation to reuse and extend the behavior of a class. You do this by writing a new class (the *delegator*) that incorporates the functionality of the original class by using an instance of the original class (the *delegatee*) and calling its methods.

Figure 4.4 shows that a class in a `Delegator` role uses a class in the `Delegatee` role.

Delegation is more general purpose than inheritance. Any extension to a class that can be accomplished by inheritance can also be accomplished by delegation.

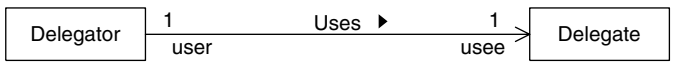

FIGURE 4.4 Delegation.

IMPLEMENTATION

The implementation of delegation is very straightforward, for it simply involves the acquisition of a reference to an instance of the class to which you want to delegate and call its methods.

The best way to ensure that a delegation is easy to maintain is to make its structure and purpose explicit. One way to do this is to make the delegation through an interface using the Interface pattern.

CONSEQUENCES

Delegation can be used without the problems that accompany inheritance. Delegation allows behavior to be easily composed at runtime.

The main disadvantage is that delegation is less structured than inheritance. Relationships between classes built by using delegation are less obvious than those built by using inheritance. The following are some strategies for improving the clarity of delegation-based relationships:

- Use consistent naming schemes to refer to objects in a particular role. For example, if multiple classes delegate the creation of widget objects, the role of the delegatee object becomes more obvious if all of the classes delegating that operation refer to delegatee objects through a variable called `widgetFactory`.
- You can clarify the purpose of a delegation by writing comments.
- Follow the Law of Demeter pattern (described in *Patterns in Java, Volume 2*), which says that if without a delegation a class would have only an indirect association with another class, the delegation should be indirect. Do not directly delegate behavior to an indirectly associated class that provides the behavior; instead, delegate it to a directly associated class and have that class delegate the behavior to the class that provides the behavior. Doing so simplifies the overall design by minimizing the number of associations between objects. In extreme cases, however, these indirect delegations can make an intermediate class less coherent by adding methods unrelated to the class's purpose. In such cases, refactor the unrelated words into a separate class by using the Pure Fabrication pattern (also described in *Patterns in Java, Volume 2*).
- Use well-known design and coding patterns. A person reading code that uses delegation will be more likely to understand the role that the objects play if the roles are part of a well-known pattern or a pattern that recurs frequently in your program.

It is not only possible but advantageous to use all three strategies at the same time.

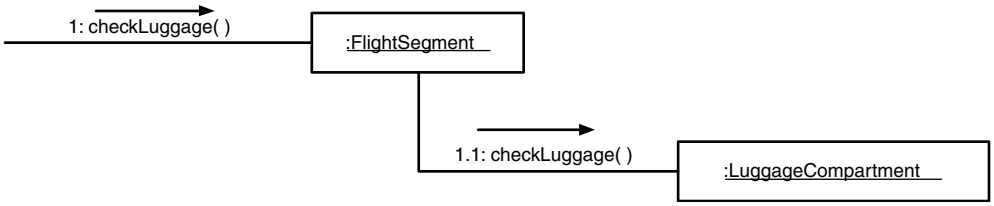

FIGURE 4.5 Check luggage.

JAVA API USAGE

The Java API is full of examples of delegation. It is the basis for Java's delegation event model in which event source objects send events to event listener objects. Event source objects do not generally decide what to do with an event; instead, they delegate the responsibility of processing the event to listener objects.

CODE EXAMPLE

For an example of delegation, we look at another part of the airline reservation system. Suppose that the system is responsible for tracking checked pieces of luggage. We can expect this part of the system to include classes to represent a flight segment,[1] a luggage compartment, and pieces of luggage, as shown in Figure 4.5.

In Figure 4.5, the `FlightSegment` class has a method called `checkLuggage` that checks a piece of luggage onto a flight. The flight class delegates that operation to an instance of the `LuggageCompartment` class.

Another common use for delegation is to implement a collection. Consider the diagram in Figure 4.6.

The `LuggageCompartment` class maintains a collection of other objects. Problem-domain classes that are responsible for maintaining a collection of other objects normally delegate the responsibility for the collection to another object, such as an instance of `java.util.ArrayList`. Implementing a collection by delegation is so common that the separate collection class is usually omitted from design drawings.

What follows are code fragments that implement the design shown in Figure 4.6. Shown first is the `FlightSegment` class that delegates the `checkLuggage` operation to the `LuggageCompartment` class:

[1] A flight segment is a portion of a trip that you take on an airline without changing planes.

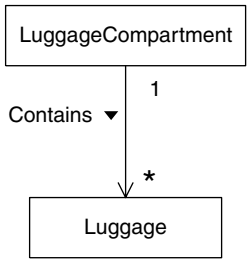

FIGURE 4.6 Luggage compartment.

```
class FlightSegment {
...
    LuggageCompartment luggage;
...
    void checkLuggage(Luggage piece) throws LuggageException {
        luggage.checkLuggage(piece);
    } // checkLuggage(Luggage)
} // class FlightSegment
```

Next, we show the `LuggageCompartment` class that delegates the collection of pieces of luggage to the `ArrayList` class:

```
class LuggageCompartment {
...
    // The pieces of luggage in this LuggageCompartment
    private ArrayList pieces = new ArrayList();
...
    void checkLuggage(Luggage piece) throws LuggageException {
...
        pieces.add(piece);
    } // checkLuggage(Luggage)
} // class LuggageCompartment
```

RELATED PATTERNS

Almost every other pattern uses delegation. Some of the patterns that rely most clearly on delegation are the Decorator pattern and the Proxy pattern.

In addition, the Interface pattern can be useful in making the structure and motivation for a delegation explicit and easier for maintainers to understand.

Interface

SYNOPSIS

Instances of a class provide data and services to instances of other classes. You want to keep client classes independent of specific data-and-service-providing classes so you can substitute another data-and-service-providing class with minimal impact on client classes. You accomplish this by having other classes access the data and services through an interface.

CONTEXT

Suppose you are writing an application to manage the purchase of goods for a business. Among the entities your program needs to know about are vendors, freight companies, receiving locations, and billing locations. One common aspect of these entities is that they all have street addresses. These street addresses appear in different parts of the user interface. You want to have a class for displaying and editing street addresses so that you can reuse it wherever there is a street address in the user interface. We call this class `AddressPanel`.

You want `AddressPanel` objects to get and set address information in a separate data object. This raises the question of what the `AddressPanel` class assumes about the class of data objects with which it will work. Clearly, you will use different classes to represent vendors, freight companies, and the like.

You can solve the problem by creating an address interface. Instances of the `AddressPanel` class would then simply require the data objects that they work with to implement the address interface. They would call the accessor methods of the interface to get and set the object's address information.

By using the indirection that the interface provides, clients of the `AddressPanel` interface are able to call the methods of a data object without having to be aware of what class it belongs to. Figure 4.7 is a class diagram showing these relationships.

FORCES

☺ An object relies on another object for data or services. If the object must assume that the other object upon which it relies belongs to a

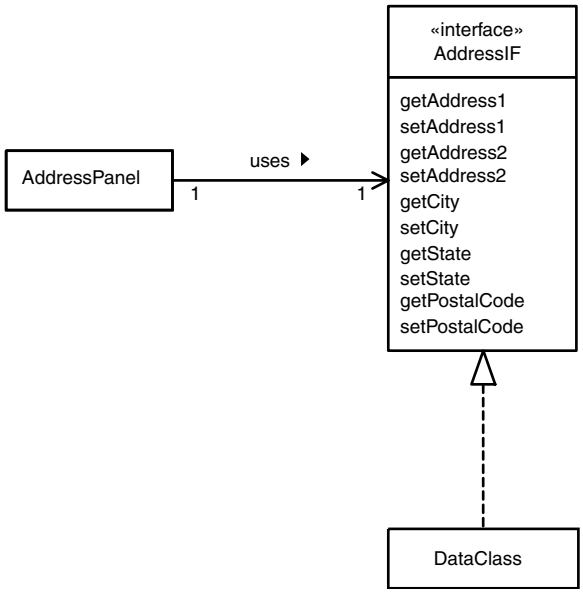

FIGURE 4.7 Indirection through address Interface.

particular class, the reusability of the object's class would be compromised.

☺ You want to vary the kind of object used by other objects for a particular purpose without making the object dependent on any class other than its own.

☹ A class's constructors cannot be accessed through an interface, because Java's interfaces cannot have constructors.

SOLUTION

To avoid classes having to depend on other classes because of a uses/used-by relationship, make the usage indirect through an interface. Figure 4.8 shows this relationship.

The following are the roles that these classes and interfaces play:

Client. The `Client` class uses classes that implement the `IndirectionIF` interface.

IndirectionIF. The `IndirectionIF` interface provides indirection that keeps the `Client` class independent of the class that is playing the `Service` role. Interfaces in this role are generally public.

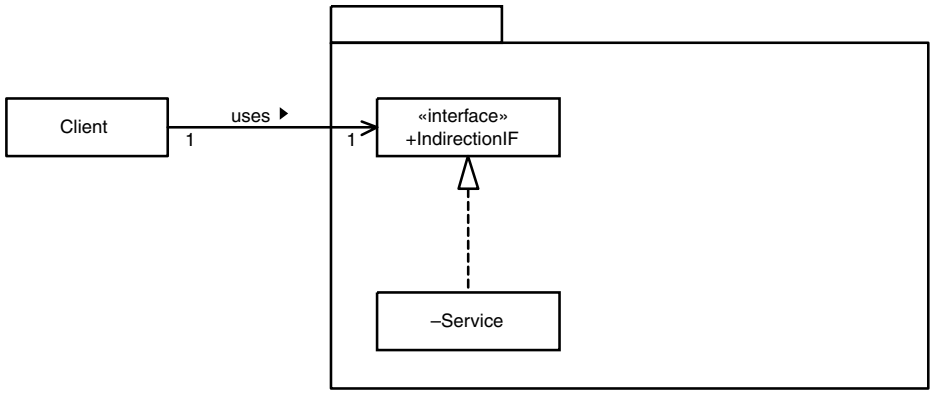

FIGURE 4.8 Interface pattern.

Service. Classes in this role provide a service to classes in the `Client` role. Classes in this role are ideally private to their package. Making `Service` classes private forces classes outside of their package to go through the interface. However, it is common to have implementations of an interface that are in different packages.

IMPLEMENTATION

Implementation of the Interface pattern is straightforward. Define an interface to provide a service, write client classes to access the service through the interface, and write service-providing classes that implement the interface.

Java interfaces cannot have constructors. For this reason, interfaces are not helpful in keeping a class responsible for creating objects independent of the class of objects that it creates. The Java API includes a class called `java.lang.reflect.Constructor` that can be used to construct objects without knowing what class they will be an instance of.

CONSEQUENCES

- ☺ Applying the Interface pattern keeps a class that needs a service from another class from being coupled to any specific class.
- ☹ Like any other indirection, the Interface pattern can make a program more difficult to understand.

JAVA API USAGE

The Java API defines the interface `java.io.FilenameFilter`. This interface declares a method called `accept`. The `accept` method takes an argument that is a filename. The method is supposed to return true or false to indicate whether the named file should be included in a collection. The Java API also provides the `java.awt.FileDialog` class that can use a `FilenameFilter` object to filter the files that it displays.

CODE EXAMPLE

The example for the Interface pattern is the `AddressPanel` class. The `AddressIF` interface is discussed under the Context heading. The following is code for the `AddressPanel` class:

```
class AddressPanel extends Panel {
    private AddressIF data;     // Data object
...
    /**
     * Set the data object that this panel will work with.
     */
    public void setData(AddressIF address) {
...
    } // setData(AddressIF)

    /**
     * Save the contents of TextFields into the data object.
     */
    public void save() {
        if (data != null) {
            data.setAddress1(address1Field.getText());
...
            data.setPostalCode(postalCodeField.getText());
        } // if data
    } // save()
} // class AddressPanel
```

Notice that the impact of the Interface pattern on the `AddressPanel` class is very small. The only difference it makes is that the type of its `data` instance variable is an interface type rather than a class.

The heart of the Interface pattern is the interface that provides the indirection between the client class and the service class. What follows is the code for the `AddressIF` interface that provides indirection for the `AddressPanel` class:

```
public interface AddressIF {
    public String getAddress1();
    public void setAddress1(String address1);
...
    public String getPostalCode() ;
    public void setPostalCode(String PostalCode);
} // interface AddressIF
```

The interface simply declares the methods required for the needed service.

What follows is code for the service class. The only impact that the Interface pattern has on the class is that it implements the `AddressIF` interface.

```
class ReceivingLocation extends Facility implements AddressIF{
    private String address1;
...
    private String postalCode;
...
    public String getAddress1() { return address1; }
    public void setAddress1(String address1) {
        this.address1 = address1;
    } // setAddress1(String)
...
    public String getPostalCode() { return postalCode; }
    public void setPostalCode(String postalCode) {
        this.postalCode = postalCode;
    } // setPostalCode(String)
} // class ReceivingLocation
```

RELATED PATTERNS

Delegation. The Delegation and Interface patterns are often used together.

Adapter. The Adapter pattern allows objects that expect another object to implement a particular interface to work with objects that don't implement the expected interface.

Strategy. The Strategy pattern uses the Interface pattern.

Anonymous Adapter. The Anonymous Adapter pattern (described in *Patterns in Java, Volume 2*) uses the Interface pattern.

Many other patterns use the Interface pattern.

Abstract Superclass

This pattern was originally described in [Rhiel00].

SYNOPSIS

Ensure consistent behavior of conceptually related classes by giving them a common abstract superclass.

CONTEXT

You want to write classes to provide sequential and read-only access to some data structures. You decide that these classes will implement the interface `java.util.Iterator`.

The `Iterator` interface includes a method called `remove`. The documented purpose of the `remove` method is to remove objects from the source over which an `Iterator` object iterates. However, the description of the `remove` method also says that it is an optional method; an implementation of the method may simply throw an `UnsupportedOperationException`.

Because your intention is to provide read-only access to data structures, you want all of your classes to implement the `remove` method by throwing an `UnsupportedOperationException`. To ensure that these classes implement the `remove` method in the same way, you create a common abstract class for all of your `Iterator` classes to inherit from. The common superclass implements the `remove` method by having it always throw an `UnsupportedOperationException`. This organization is shown in Figure 4.9.

FORCES

- ☺ You want to ensure that logic common to related classes is implemented consistently for each class.
- ☺ You want to avoid the runtime and maintenance overhead of redundant code.
- ☺ You want to make it easy to write related classes.
- ☹ You want to organize common behavior, although in many situations, inheritance is not an appropriate way to accomplish this. The Delegation pattern describes this in detail.

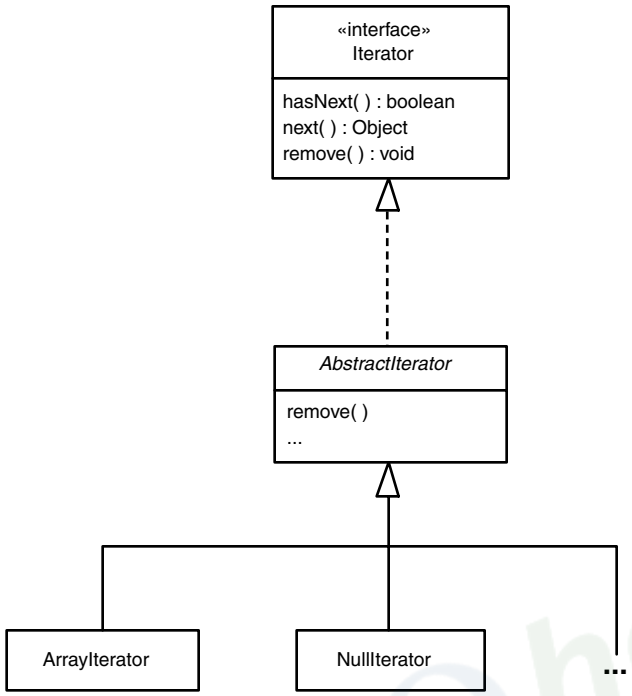

FIGURE 4.9 Iterators with abstract superclass.

SOLUTION

Organize the common behavior of related classes into an abstract superclass.

To the extent possible, organize variant behavior into methods with common signatures.[2] Declare the abstract superclass to have abstract methods with these common signatures. Figure 4.10 shows this organization.

The following are the roles that classes play in the Abstract Superclass pattern:

AbstractSuperclass. A class in this role is an abstract superclass that encapsulates the common logic for related classes. The related classes extend this class so they can inherit methods from it. Methods whose signature and logic are common to the related classes are put into the superclass so their logic can be inherited by the related classes that extend the superclass.

[2] A method's signature is the combination of its name and formal parameters.

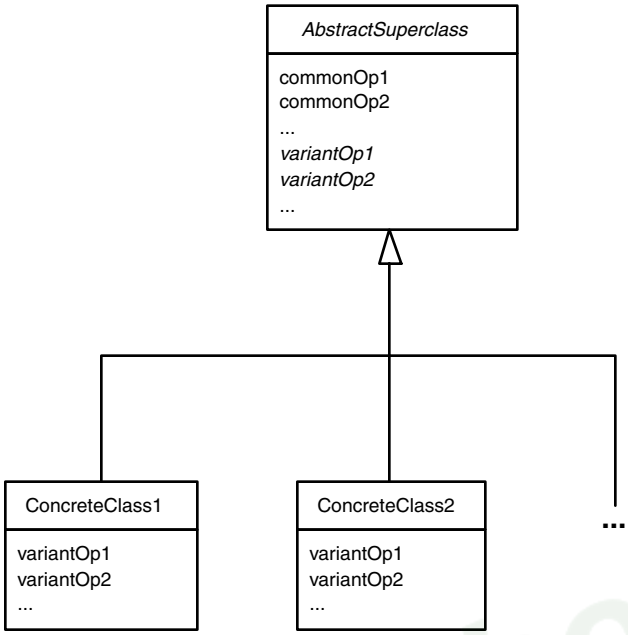

FIGURE 4.10 Abstract Superclass pattern.

Methods with different logic but the same signature are declared in the abstract class as abstract methods, ensuring that each concrete subclass has a method with those signatures.

ConcreteClass1, ConcreteClass2, and so on. A class in this role is a concrete class whose logic and purpose is related to other concrete classes. Methods common to these related classes are refactored into the abstract superclass.

Common logic that is not encapsulated in common methods is refactored into common methods.

IMPLEMENTATION

If the common method signatures are public, they should, if possible, be organized into a Java interface that the abstract class implements.

CONSEQUENCES

☺ Fewer test cases may be needed to completely test your classes, because there are fewer pieces of code to test.

☹ Using the Abstract Superclass pattern creates dependencies between the superclass and its subclasses. Changes to the superclass may have unintended effects on some subclasses, thus making the program harder to maintain.

JAVA API USAGE

The class `java.awt.AWTEvent` is an abstract class for classes that encapsulate events related to the graphical user interface (GUI). It defines a small number of methods that are common to user-interface event classes.

CODE EXAMPLE

Implementations of the classes discussed under the Context heading are the code example for this pattern. These classes are taken from the ClickBlocks software, which you can find on the Web site for this book in the `org.clickblocks.util` package.

The following is a listing of the `AbstractIterator` class:

```
/**
 * This abstract class provides the convenience of allowing
 * Iterators to be defined by overriding just the
 * getNextElement single method.
*/
abstract public class AbstractIterator implements Iterator {
    private Object nextElement;

    /**
     * This method must be called by a subclass's constructor.
     */
    protected void init() {
        nextElement = getNextElement();
    } // init()

    /**
     * This method returns the next element in the data
     * structure to be traversed.  If there is no next
     * element, then return this object.
     */
    public abstract Object getNextElement();

    /**
     * Return true if the iteration has more elements.
     */
    public boolean hasNext(){
        return nextElement!=this;
    } // hasNext()
```

```
    /**
     * Returns the next element in the iteration.
     *
     * @exception NoSuchElementException iteration has no more elements.
     */
    public Object next(){
        if (nextElement==this) {
            throw new NoSuchElementException();
        } // if
        Object previous = nextElement;
        nextElement = getNextElement();
        return previous;
    } // next()

    /**
     * Remove from the underlying collection the last element
     * returned by the next method.
     *
     * @exception UnsupportedOperationException
     *            if the remove operation is not supported by
     *            this Iterator.
     */
    public void remove(){
        throw new UnsupportedOperationException();
    } // remove()
} // class AbstractIterator
```

RELATED PATTERNS

Interface and Abstract Class. The Interface and Abstract Class pattern uses the Abstract Superclass pattern.

Template Method. The Template Method pattern uses the Abstract Superclass pattern.

Interface and Abstract Class

SYNOPSIS

You need to keep client classes independent of classes that implement a behavior and ensure consistency of behavior between the behavior-implementing classes. Don't choose between using an interface and an abstract class; have the classes implement an interface *and* extend an abstract class.

CONTEXT

You are designing a framework. You want to hide the class or classes that implement some behavior by making the classes private to their package and having them implement a public interface. For the sake of consistency and convenience of implementation, you want the classes to extend a common abstract class. You are not sure how to decide between basing the classes on an interface and an abstract class.

FORCES

- ☺ Using the Interface pattern, Java interfaces can be used to hide the specific class that implements a behavior from the clients of the class.
- ☺ Organizing classes that provide related behaviors with a common superclass helps to ensure consistency of implementation. Through reuse, it may also reduce the effort required for implementation.
- ☹ When people are presented with two different ways to improve the organization of classes, there is a common tendency to choose either one or the other.

SOLUTION

If you are presented with the need to hide from its clients the class of an object that provides a service, then use the Interface pattern. Have the client objects access the service-providing object indirectly through an interface. The indirection allows the clients to access the service-providing object without having to know what kind of objects they are.

If you need to design a set of related classes that provide similar functionality, then organize the common portions of their implementation into an abstract superclass.

If you are presented with both of these needs in the same object design, then use both an interface and an abstract class as shown in Figure 4.11.

When using the combination of an interface and abstract class in this way, the interface is public; the abstract class is package private, if possible.

CONSEQUENCES

☺ Using the Interface and Abstract Class pattern allows an object design to benefit from both an interface and an abstract class.

JAVA API USAGE

The package `javax.swing.table` contains interfaces and classes to support tables in a user interface. For every table that appears in a user interface, a corresponding data model object contains the data values displayed in the table. To be used as the data model for a table, an object must be an

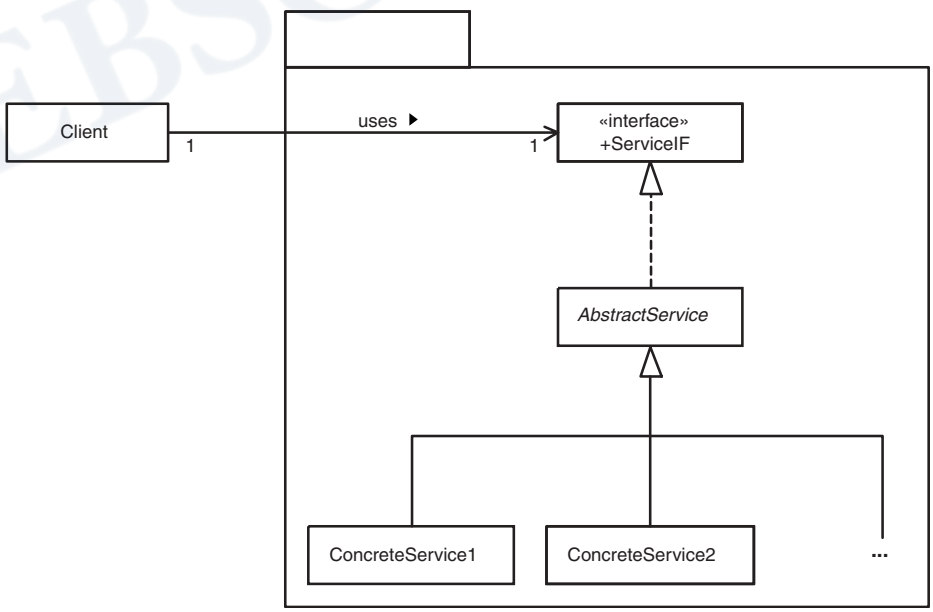

FIGURE 4.11 Interface and abstract class.

instance of a class that implements the `javax.swing.table.TableModel` interface. The package also includes a class named `AbstractTableModel`. `AbstractTableModel` is an abstract class that contains some default logic useful in implementing the methods declared by the `TableModel` interface. Finally, there is a concrete class named `DefaultTableModel` that is the default class used to instantiate a data model for a table. These relationships are shown in Figure 4.12.

CODE EXAMPLE

The code example for the Interface and Abstract Class pattern consists of an interface and classes for managing a data structure called a *doubly linked list*. There is a class called `DoubleLinkedListMgr` that performs various insert and delete operations on the elements of a doubly linked list. The `DoubleLinkedListMgr` class does not require that the objects in the doubly linked list be instances of any particular class. It does require that all of the elements implement an interface called `DoubleLinkIF`. There is an abstract class called `AbstractDoubleLink` that implements the `DoubleLinkIF` interface. Extending the `AbstractDoubleLink` class is a convenient way to write concrete classes that can be manipulated in a doubly linked list.

These classes and the interface are part of `org.clickblocks.dataStructure` package of the ClickBlocks software on the Web site for this book.

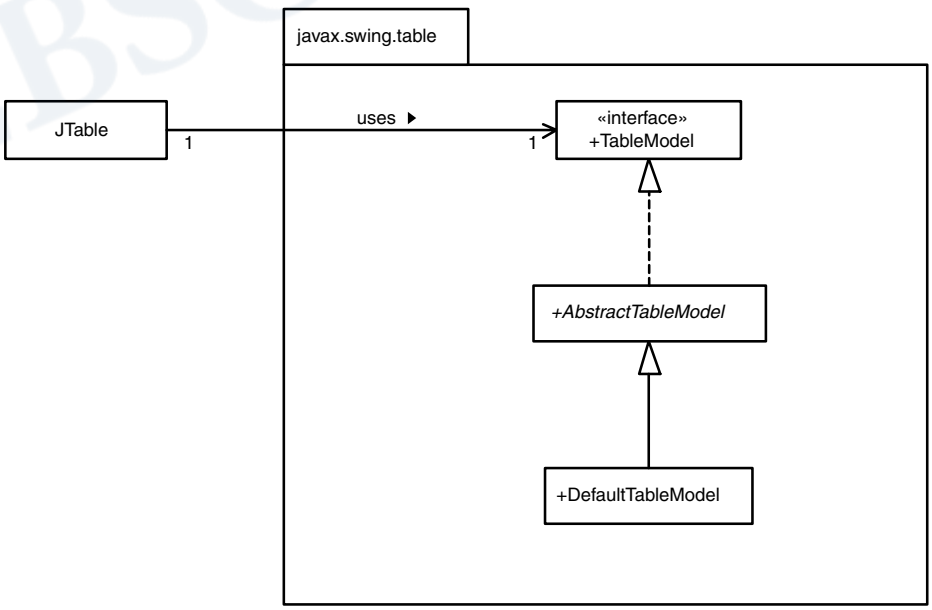

FIGURE 4.12 javax.swing.table relationships.

Doubly Linked List

A doubly linked list is a data structure organized into a sequence so that each element contains a reference to its successor and predecessor. Figure 4.13 shows an example of this structure.

The advantage of a doubly linked list over an array is the amount of work it takes to insert and delete objects. When you insert or delete objects in an array, you must shift everything in the array after the insertion or deletion point. The larger the array, the longer the average insertion and deletion will take. Inserting or deleting objects in a doubly linked list involves adjusting predecessor and successor references.

Inserting or deleting an element in a doubly linked list always takes the same amount of time, no matter how many elements there are. The drawback is that finding the nth element of a doubly linked list requires a search through the first n elements in the list. The larger n is, the longer this takes. Indexing the nth element in an array takes the same amount of time, no matter how large the array is.

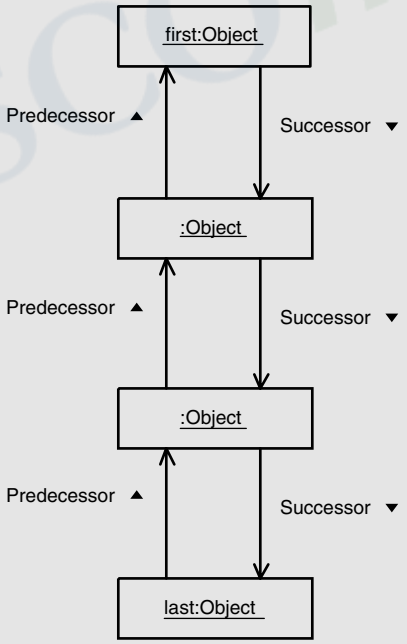

FIGURE 4.13 Doubly linked list.

The following is a listing of the `DoubleLinkIF` interface:

```
public interface DoubleLinkIF {
    /**
     * Return the node that follows this one in the linked
     * list or null if this is the last node.
     */
    public DoubleLinkIF getNext() ;

    /**
     * Set the node that is to follow this one in the linked
     * list.
     *
     * @param node
     *        The node that is to follow this one in the
     *        linked list or null if this node is to be the
     *        last one in the list.
     */
    public void setNext(DoubleLinkIF newValue) ;

    /**
     * Return the node that precedes this one in the linked
     * list or null if this is the first node.
     */
    public DoubleLinkIF getPrev() ;

    /**
     * Set the node that is to precede this one in the linked
     * list.
     *
     * @param node
     *        The node that is to precede this one in the
     *        linked list or null if this node is to be the
     *        first one in the list.
     */
    public void setPrev(DoubleLinkIF newValue) ;
} // interface DoubleLinkIF
```

The following is a listing of the abstract class `AbstractDoubleLink` that implements the `DoubleLinkIF` interface:

```
public abstract class AbstractDoubleLink
                implements DoubleLinkIF {
    private DoubleLinkIF previous;
    private DoubleLinkIF next;

    /**
     * Return the node that follows this one in the linked
     * list or null if this is the last node.
     */
    public DoubleLinkIF getNext() { return next; }
```

```
    /**
     * Set the node to follow this one in the linked list.
     *
     * @param node
     *        The node to follow this one or null if this node
     *        is to be the last one in the list.
     */
    public void setNext(DoubleLinkIF newValue) {
        next = newValue;
    } // setNext(DoubleLinkIF)

    /**
     * Return the node that precedes this one or null
     * if this is the first node.
     */
    public DoubleLinkIF getPrev() { return previous; }

    /**
     * Set the node that is to precede this one.
     *
     * @param node
     *        The node that is to precede this one or
     *        null if this node is to be the first.
     */
    public void setPrev(DoubleLinkIF newValue) {
        previous = newValue;
    } // setPrev(DoubleLinkIF)
...
} // class AbstractDoubleLink
```

Classes that extend the `AbstractDoubleLink` class generally add additional information of their own.

RELATED PATTERNS

Interface. The Interface and Abstract Class pattern uses the Interface pattern.

Abstract Class. The Interface and Abstract Class pattern uses the Abstract Superclass pattern.

Immutable

The Immutable pattern is fundamental in a different sense than other patterns presented in this chapter. The Immutable pattern is considered fundamental because the more appropriate places you use it, the more robust and maintainable your programs will be.

SYNOPSIS

The Immutable pattern increases the robustness of objects that share references to the same object and reduces the overhead of concurrent access to an object. It achieves this by not allowing the shared object's contents to change after the object is constructed. The Immutable pattern also avoids the need to synchronize multiple threads of execution that share an object.

CONTEXT

Value objects are objects whose primary purpose is to encapsulate values rather than to provide behavior. For example, the class `java.awt.Rectangle` encapsulates the position and dimensions of a rectangle.

In situations where multiple objects share access to the same value object, a problem can arise if changes to the shared object are not properly coordinated between the objects that share it. This coordination can require careful programming that is easy to get wrong. If the changes to and fetches of the shared objects' state are done asynchronously, then in addition to the greater likelihood of bugs, correctly functioning code will have the overhead of synchronizing the accesses to the shared objects' state.

The Immutable pattern avoids these problems. It organizes a class so that the state information of its instances never changes after they are constructed.

Suppose you are designing a multiplayer game program that involves the placement and occasional movement of objects on a playing field. In the course of designing classes for the program, you decide to use immutable objects to represent the position of objects on the playing field. An organization of a class for modeling position is shown in Figure 4.14.

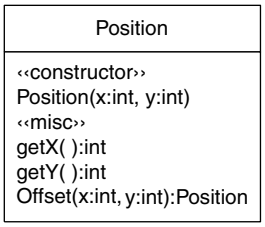

FIGURE 4.14 Immutable position.

You have a class called `Position` that has an *x* and a *y* value associated with its instances. The class has a constructor that specifies the *x* and *y* values. It also has methods to fetch the *x* and *y* values associated with its instances, as well as a method that creates a new `Position` object at a given *x* and *y* offset from an existing position. It does not have any methods to modify its *x* or *y* values. If an object's position changes, it will be made to refer to a new position object.

FORCES

- ☺ Your program uses instances of a class that is passive in nature. The instances do not ever need to change their own state. The instances of that class are used by multiple other objects.
- ☺ Coordinating changes to the contents of a value object used by multiple objects can be a source of bugs. When the contents of a value object change, all the objects that use it may need to be informed. Also, when multiple objects use an object, they may attempt to change its state in inconsistent ways.
- ☺ If multiple threads modify the contents of a value object, the modification operations must be synchronized to ensure the consistency of the contents. The overhead of synchronizing the threads may add an unacceptable overhead to accessing the value object's contents.
- ☺ An alternative to modifying the contents of a value object is to replace the entire object with another object that has different contents. Doing so avoids the need for synchronization among threads that only fetch the contents of the value object.
- ☹ If there are multiple threads that update the contents of an object, replacing the object instead of updating its contents will not avoid the need for synchronizing the threads that do the updating.
- ☹ Replacing a value object with a new value object that contains an updated version of the values in the old object involves copying unchanged values from the old object to the new. If the changes to an object are frequent, or if an object has a large amount of state information associated with it, the cost of replacing value objects may be prohibitive.

SOLUTION

To avoid having to manage the synchronization of changes to value objects used by multiple other objects, make the shared objects immutable, disallowing any changes to their state after they are constructed. You can accomplish this task by not including any methods, other than constructors, in their class that modify state information. The organization of such a class is shown in Figure 4.15.

Notice that the class has accessor methods to get state information but none to set it.

IMPLEMENTATION

There are two concerns when implementing the Immutable pattern:

- No method, other than a constructor, should modify the values of a class's instance variables.
- Any method that computes new state information must store the information in a new instance of the same class rather than modifying the existing object's state.

One possible unexpected detail of implementing the Immutable pattern is that it usually does not involve declaring variables with the `final` modifier. The values of final instance variables are normally provided from within their class. However, the values of an immutable object's instance variables normally are provided by another class that instantiates the object.

CONSEQUENCES

☺ Since the state of immutable objects never changes, there is no need to write code to manage such changes.

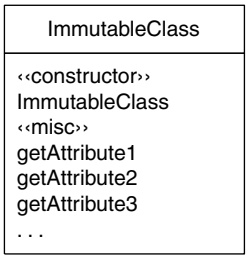

FIGURE 4.15 Immutable pattern.

- ☺ An immutable object is often used as the value of another object's attribute. If the value of an object's attribute is an immutable object, then it may not be necessary for access to the value's attribute to be synchronized. The reason is that Java guarantees that assigning an object reference to a variable is always done as an atomic operation. If the value of a variable is an object reference and one thread updates its value and another fetches its value, the other thread will fetch either the new or the old object reference.
- ☹ Operations that would otherwise have changed the state of an object must create a new object. This is an overhead that mutable objects do not incur.

JAVA API USAGE

Instances of the `String` class are immutable. The sequence of characters that a `String` object represents is determined when it is constructed. The `String` class does not provide any methods for changing the sequence of characters that a `String` object represents. Methods of the `String` class, such as `toLowerCase` and `substring`, compute a new sequence of characters and then return the new sequence of characters in a new `String` object.

CODE EXAMPLE

The following is what the code for the `Position` class described in the Context section might look like:

```
class Position {
    private int x;
    private int y;

    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    } // constructor(int, int)

    public int getX() { return x; }

    public int getY() { return y; }

    public Position offset(int xOffset, int yOffset) {
        return new Position(x+xOffset, y+yOffset);
    } // offset(int, int)
} // class Position
```

RELATED PATTERNS

Single Threaded Execution. The Single Threaded Execution pattern is the pattern most frequently used to coordinate access by multiple threads to a shared object. The Immutable Object pattern can be used to avoid the need for the Single Threaded Execution pattern or any other kind of access coordination.

Read-Only Interface. The Read-Only Interface pattern is an alternative to the Immutable Object pattern. It allows some objects to modify a value object while other objects can only fetch its values.

Marker Interface

The Marker Interface pattern occurs rarely outside of utility classes. However, it is included in this chapter because it takes advantage of the fundamental nature of class declarations.

SYNOPSIS

The Marker Interface pattern uses the fact that a class implements an interface to indicate semantic Boolean attributes of the class. It works particularly well with utility classes that must determine something about objects without assuming that they are an instance of any particular class.

CONTEXT

Java's `Object` class defines a method called `equals`. The argument to `equals` can be a reference to any object. Since Java's `Object` class is the ultimate superclass of all other classes in Java, all other classes inherit the `equals` method from the `Object` class. The implementation of `equals` provided by the `Object` class is equivalent to the `==` operator. It returns true if the object passed to it is the same object as the object it is associated with. Classes that want their instances to be considered equal if they contain the same values override the `equals` method appropriately.

Container objects, such as `java.util.ArrayList`, call an object's `equals` method when performing a search of their contents to find an object that is equal to a given object. Such searches might call an object's `equals` method for each object in the container objects. This is wasteful in those cases where the object being searched for belongs to a class that does not override the `equals` method. It is faster to use the `==` operator to determine whether two objects are the same object than it is to call the `Object` class's implementation of the `equals` method. If the container class were able to determine that the object being searched for belongs to a class that does not override the `equals` method, then it could use the `==` operator instead of calling `equals`. The problem, however, is that no quick way exists to determine whether an arbitrary object's class has overridden the `equals` method.

It is possible to provide a hint to container classes to let them know that it is correct to use the `==` operator for an equality test on instances of a

class. You can define an interface called `EqualByIdentity` that declares no methods or variables. You can then write container classes to assume that if a class implements `EqualByIdentity`, its equality comparison can be done by using the `==` operator.

An interface that does not declare methods or variables and is used to indicate attributes of classes that implement them is said to be a marker interface.

FORCES

- ☺ Utility classes may need to know something about the intended use of an object's class that is either true or false without relying on objects being an instance of a particular class.
- ☺ Classes can implement any number of interfaces.
- ☺ It is possible to determine whether an object's class implements a known interface without relying on the object being an instance of any particular class.
- ☹ Some attributes about the intended use of a class may change during the class's lifetime.

SOLUTION

For instances of a utility class to determine whether another class's instances are included in a classification without the utility class having knowledge of other classes, the utility class can determine whether other classes implement a marker interface. A marker interface is an interface that does not declare any methods or variables. You declare a class to implement a marker interface to indicate that it belongs to the classification associated with the marker interface. Figure 4.16 shows these relationships.

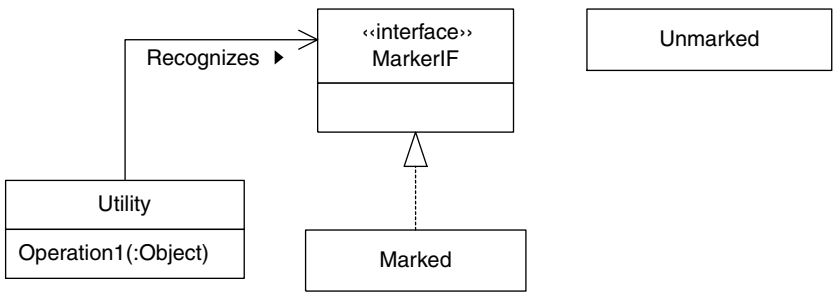

FIGURE 4.16 Marker Interface class diagram.

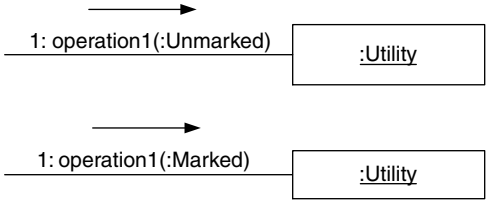

FIGURE 4.17 Marker Interface collaboration.

Figure 4.16 shows a marker interface called `MarkerIF`. Figure 4.16 also shows a class called `Marked` that implements `MarkerIF` and a class called `Unmarked` that doesn't. There is also a utility class called `Utility` that is aware of the `MarkerIF` interface. The collaboration between these classes is shown in Figure 4.17.

Instances of `UtilityClass` receive calls to their `operation1` method. The parameter passed to that method can be an object that implements or does not implement `MarkerIF`.

IMPLEMENTATION

The essence of the Marker Interface pattern is that an object that either does or does not implement a marker interface is passed to a method of a utility class. The formal parameter that corresponds to the object is typically declared as `Object`. If appropriate, it is reasonable to declare that formal parameter to be a more specialized class.

It is also possible to use an interface that declares methods in the Marker Interface pattern. In such cases, the interface used as a marker interface usually extends a pure marker interface.

Declaring that a class implements a marker interface implies that the class is included in the classification implied by the interface. It also implies that all subclasses of that class are included in the classification. If there is any possibility that someone will declare a subclass that does not fit the classification, then you should take measures to prevent that from happening. Such measures might include declaring the class final to prevent it from being subclassed or declaring its `equals` method final to prevent it from being overridden.

CONSEQUENCES

☺ Instances of utility classes are able to make inferences about objects passed to their methods without depending on the objects to be instances of any particular class.

- The relationship between the utility class and the marker interface is transparent to all other classes except those that implement the interface.

JAVA API USAGE

A class indicates that its instances may be serialized by implementing the `Serializable` interface. Instances of the `ObjectOutputStream` class write objects as a stream of bytes. An instance of the `ObjectInputStream` class can read the stream of bytes and turn it back into an object. The conversion of an object to a stream of bytes is called *serialization.* There are many reasons why instances of some classes should not be serialized. The `ObjectOutputStream` class refuses to serialize objects unless their class implements the `Serializable` interface to indicate that its serialization is allowed.

CODE EXAMPLE

For an example of an application of the Marker Interface pattern, see the following class that implements a linked-list data structure. At the bottom of the listing are methods called `find`, `findEq`, and `findEquals`. The purpose of all three methods is to find a `LinkedList` node that refers to a specified object. The `find` method is the only one of the three that is public. The `findEq` method performs the necessary equality tests by using the `==` operator. The `findEquals` method performs the necessary equality tests by using the `equals` method of the object being searched for. The `find` method decides whether to call the `findEq` method or the `findEquals` method by determining whether the object to search for implements the marker interface `EqualByIdentity`.

```
public class LinkedList implements Cloneable, java.io.Serializable {
...
    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is normally determined by calling the given
     * object's equals method.  However, if the given object implements
     * the EqualByIdentity interface, then equality will be determined
     * by the == operator.
     */
    public LinkedList find(Object target) {
        if (target == null || target instanceof EqualByIdentity)
          return findEq(target);
        else
          return findEquals(target);
    } // find(Object)
```

```
    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is determined by the == operator.
     */
    private synchronized LinkedList findEq(Object target) {
...
    } // find(Object)

    /**
     * Find an object in a linked list that is equal to the given
     * object. Equality is determined by calling the given
     * object's equals method.
     */
    private synchronized LinkedList findEquals(Object target) {
...
    } // find(Object)
} // class LinkedList
```

RELATED PATTERNS

Snapshot. The Marker Interface pattern is used as part of the Snapshot pattern to allow serialization of objects.

Polymorphism. The Polymorphism pattern (described in *Patterns in Java, Volume 2*) describes the alternative way to vary the behavior of a method call.

Proxy

Proxy is a very general pattern that occurs in many other patterns but never by itself in its pure form. The Proxy pattern was described previously in [GoF95].

SYNOPSIS

The Proxy pattern forces method calls to an object to occur indirectly through a proxy object that acts as a surrogate for the other object, delegating method calls to that object. Classes for proxy objects are declared in a way that usually eliminates the client object's awareness that it is dealing with a proxy.

CONTEXT

A proxy object is an object that receives method calls on behalf of another object. Client objects call the proxy object's method. The proxy object's methods do not directly provide the service that its clients expect; instead, they call the methods of the object that provides the actual service. Figure 4.18 shows this structure.

Although a proxy object's methods do not directly provide the service its clients expect, the proxy object provides some management of those services. Proxy objects share a common interface with the service-providing object. Whether client objects directly access a service-providing object or a proxy object, they access it through the common interface rather than an instance of a particular class. Doing so allows client objects to be unaware that they call the methods of a proxy object rather than the methods of the actual service-providing object. Transparent management of another object's services is the basic reason for using a proxy object.

A proxy object can be used to provide many types of service management. Some of the more important types are documented elsewhere in this

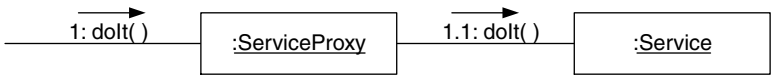

FIGURE 4.18 Method calls through a proxy object.

work as patterns in their own right. The following are some of the more common uses for proxies:

- Create the appearance that a method that takes a long time to complete returns immediately.
- Create the illusion that an object on a different machine is an ordinary local object. This kind of proxy—called a *remote proxy,* or *stub*—is used by RMI, CORBA and other ORBs (object request brokers). Stub classes are described as part of the discussion of ORBs in *Patterns in Java, Volume 3.*
- Control access to a service-providing object based on a security policy. This use of proxies is described as the Protection Proxy pattern in *Patterns in Java, Volume 3.*
- Create the illusion that a service object exists before it actually does. Doing so can be useful if a service object is expensive to create and its services may not be needed. This use of proxies is documented as the Virtual Proxy pattern.

FORCES

- ☺ It is not possible for a service-providing object to provide a service at a convenient time or place.
- ☺ Gaining visibility to an object is complex and you want to hide that complexity.
- ☺ Access to a service-providing object must be controlled without adding complexity to the service-providing object or coupling the service to the access control policy.
- ☺ The management of a service should be provided in a way that is transparent to the clients of that service.
- ☺ The clients of a service-providing object do not care about the identity of the object's class or which instance of its class they are working with.

SOLUTION

Transparent management of a service-providing object can be accomplished by forcing all access to the service-providing object to be accomplished through a proxy object. For the management to be transparent, both the proxy object and the service-providing object must either implement a common interface or be instances of a common superclass.

Figure 4.19 shows the organization of the Proxy pattern but not details for implementing any particular access management policy. The

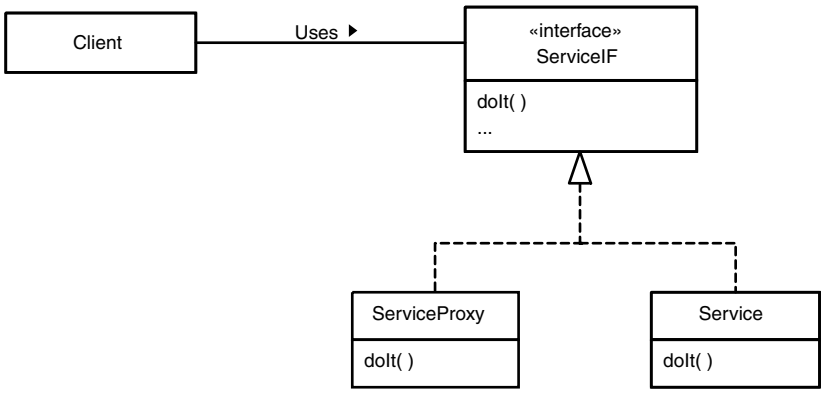

FIGURE 4.19 Proxy class diagram.

Proxy pattern is not very useful unless it implements some particular access management policy. The Proxy pattern is so commonly used with some access management policies that these combinations are described elsewhere as patterns in their own right.

IMPLEMENTATION

Without any specific management policy, the implementation of the Proxy pattern simply involves creating a class that shares a common superclass or interface with a service-providing class and delegates operations to instances of the service-providing class.

CONSEQUENCES

- ☺ The service provided by a service-providing object is managed in a manner transparent to the object and its clients.
- ☺ Unless the use of proxies introduces new failure modes, there is normally no need for the code of client classes to reflect the use of proxies.

CODE EXAMPLE

The Proxy pattern is not useful in its pure form; it must be combined with a service management behavior to accomplish anything useful. This example of the Proxy pattern uses proxies to defer an expensive operation until it is actually needed. If the operation is not needed, the operation is never performed.

The example is a proxy for instances of classes such as `java.util.HashMap` that implement both the `java.util.Map` and the `java.long.Cloneable` interfaces. The purpose of the proxy is to delay cloning the underlying `Map` object until it is known that this expensive operation is actually needed.

Clone()

All Java classes inherit the `clone` method from the class `java.lang.Object`. An object's `clone` method returns a shallow copy of the object. The object that the `clone` method returns is an instance of the same class as the original object. Its instance values all have the same values as in the original object. The instance variables of the copy refer to the same objects as the original object.

Since it is a security hole to be able to copy an object that contains sensitive information, the `clone` method inherited from the `Object` class throws an exception unless the class of the object permits cloning. A class permits its instances to be cloned if it implements the `java.lang.Cloneable` interface.

Many classes that permit their instances to be cloned override the `clone` method to avoid situations where their instances are sharing objects that they should not share.

One reason for cloning a `Map` object is to avoid holding a lock on the object for a long time when all that is desired is to fetch multiple key-value pairs. In a multi-threaded program, to ensure that a `Map` object is in a consistent state while fetching key-value pairs from it, you can use a synchronized method to obtain exclusive access to the `Map` object. While that is happening, other threads wait to gain access to the same `Map` object. Such waiting may be unacceptable

Not every class that implements the `Map` interface permits its instances to be cloned. However, many `Map` classes, such as `java.util.HashMap` and `java.util.TreeMap`, do permit their instances to be cloned.

Cloning a `Map` object prior to fetching values out of it is a defensive measure. Cloning the `Map` object avoids the need to obtain a synchronization lock on a `Hashtable` beyond the time it takes for the clone operation to complete. When you have a freshly cloned copy of a `Map` object, you can be sure that no other thread has access to the copy. Because no other thread has access to the copy, you will be able to fetch key-value pairs from the copy without any interference from other threads.

If after you clone a `Map` object no subsequent modification to the original `Map` object occurs, the time and memory spent in creating the clone was wasted. The point of this example is to avoid waste. It accomplishes this by delaying the cloning of a `Map` object until a modification to it actually occurs.

The name of the proxy class is `LazyCloneMap`. Instances of `LazyCloneMap` are a copy-on-write proxy for a `Map` object. When a proxy's clone method is called, it returns a copy of the proxy but does not copy the underlying `Map` object. At this point, both the original and the copy of the proxy refer to the same underlying `Map` object. When one of the proxies is asked to modify the underlying `Map` object, it will recognize that it uses a shared underlying `Map` object; it will clone the underlying `Map` object before it makes the modification. Figure 4.20 shows the structure of the `LazyCloneMap` class.

What follows is the beginning of a listing of the `LazyCloneMap` class.

```
public class LazyCloneMap implements Map, Cloneable {
    /**
     * The Map object that this object is a proxy for.
     */
    private Map underlyingMap;
```

Proxy classes have a way for a proxy object to refer to the object it is a proxy for. This proxy class has an instance variable that refers to the underlying `Map` object. `LazyCloneMap` objects know when they share their underlying `Map` object with other `LazyCloneMap` objects by keeping a count of how many refer to the same underlying `Map` object. They keep this count in an instance of a class named `MutableInteger`. The `MutableInteger` object is shared or not by the same `LazyCloneMap` objects that share the underlying `Map` object. The listing for `MutableInteger` appears later.

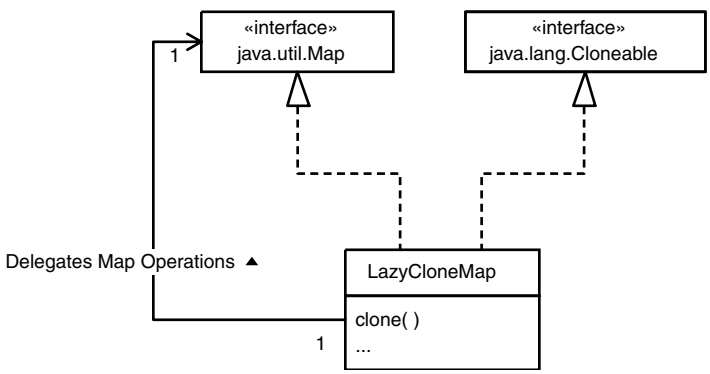

FIGURE 4.20 LazyCloneMap.

```
/**
 * This is the number of proxy objects that share the same
 * underlying map.
 */
private MutableInteger refCount;
```

The `clone` method that classes inherit from the `Object` class is `protected`. There is no interface that declares the public `clone` method implemented by classes such as `HashMap` and `TreeMap`. For this reason, the `LazyCloneMap` class must access the `clone` method of underlying `Map` objects through an explicit `java.lang.reflect.Method` object.

```
/**
 * This is used to invoke the clone method of the
 * underlying Map object.
 */
private Method cloneMethod;

private static Class[] cloneParams = new Class[0];


/**
 * Constructor
 *
 * @param underlyingMap
 *        The Map object that this object should be a
 *        proxy for.
 * @throws NoSuchMethodException
 *         If the underlyingMap object does not have a
 *         public clone() method.
 * @throws InvocationTargetException
 *         The object constructed by this constructor uses
 *         a clone of the given Map object.  If the Map
 *         object's clone method throws an exception, this
 *         constructor throws an InvocationTargetException
 *         whose getCause method returns the original *
 *         exception.
 */
public LazyCloneMap(Map underlyingMap)
                        throws NoSuchMethodException,
                               InvocationTargetException {
    Class mapClass = underlyingMap.getClass();
    cloneMethod = mapClass.getMethod("clone",cloneParams);
    try {
        this.underlyingMap =
          (Map)cloneMethod.invoke(underlyingMap, null);
    } catch (IllegalAccessException e) {
        // This should not happen.
    } // try
    refCount = new MutableInteger(1);
} // constructor(Map)
```

This class's `clone` method does not copy the underlying `Map` object. It does increment the reference count that is also shared by the original `LazyCloneMap` object and the copy. Incrementing the value allows the `LazyCloneMap` objects to know that they share their underlying `Map` object.

```
public Object clone() {
    LazyCloneMap theClone;
    try {
        Cloneable original = (Cloneable)underlyingMap;
        theClone = (LazyCloneMap)super.clone();
    } catch (CloneNotSupportedException e) {
        // this should never happen
        theClone = null;
    } // try
    refCount.setValue(1+refCount.getValue());
    return theClone;
} // clone()
```

The following private method is called by public methods of the `LazyCloneMap` class, such as `put` and `clear`, that modify the underlying `Map` object. The public methods call this private method before modifying the underlying `Map` object to ensure that they are not sharing the underlying `Map` object.

```
private void ensureUnderlyingMapNotShared() {
    if (refCount.getValue()>1) {
        try {
            underlyingMap =
              (Map)cloneMethod.invoke(underlyingMap, null);
            refCount.setValue(refCount.getValue()-1);
            refCount = new MutableInteger(1);
        } catch (IllegalAccessException e) {
            // This should not happen.
        } catch (InvocationTargetException e) {
            Throwable cause = e.getCause();
            throw new RuntimeException("clone failed",
                                       cause);
        } // try
    } // if
} // ensureUnderlyingMapNotShared()
```

The `ensureUnderlyingMapNotShared` method begins by determining whether the value of the reference count is greater than one. If it is greater than one, it will know that this `LazyCloneMap` object shares its underlying `Map` object with other `LazyCloneMap` objects. The `ensureUnderlyingMap-NotShared` method clones the underlying `Map` object so this `LazyCloneMap` object will have its own copy of the underlying `Map` object that it does not share with any other `LazyCloneMap` object. It decrements the reference

count so that the other LazyCloneMap objects with which this object shared an underlying Map object will know that they no longer share the same underlying Map object with this object. Also, it creates a new object to contain the reference count of 1 indicating that this LazyCloneMap object does not share its underlying Map object.

The rest of the methods in the LazyCloneMap class delegate their work to the corresponding method of the underlying Map object.

```
public int size(){
    return underlyingMap.size();
}

public boolean isEmpty(){
    return underlyingMap.isEmpty();
}

public boolean containsKey(Object key){
    return underlyingMap.containsKey(key);
}

public boolean containsValue(Object value){
    return underlyingMap.containsValue(value);
}

public Object get(Object key){
    return underlyingMap.get(key);
}
```

Because the next four methods modify the underlying Map object, they call the ensureUnderlyingMapNotShared method before they delegate their work to the underlying Map object.

```
public Object put(Object key, Object value){
    ensureUnderlyingMapNotShared();
    return underlyingMap.put(key, value);
}

public Object remove(Object key){
    ensureUnderlyingMapNotShared();
    return underlyingMap.remove(key);
}

public void putAll(Map m){
    ensureUnderlyingMapNotShared();
    underlyingMap.putAll(m);
}

public void clear(){
    ensureUnderlyingMapNotShared();
    underlyingMap.clear();
}
```

```
    public Set keySet(){
        return underlyingMap.keySet();
    }

    public Collection values(){
        return underlyingMap.values();
      }

    public Set entrySet(){
        return underlyingMap.entrySet();
    }

    public boolean equals(Object that){
        return underlyingMap.equals(that);
    }

    public int hashCode(){
        return underlyingMap.hashCode();
    }
}
```

The following is a listing of the `MutableInteger` class that the `LazyCloneMap` class uses:

```
public class MutableInteger {
    public int val;

    public MutableInteger( int value ) {
        setValue( value );
    }

    public int getValue() {
        return( val );
    }

    public void setValue( int value ) {
        val = value;
    }
    ...
}
```

RELATED PATTERNS

Protection Proxy. The Protection Proxy pattern (described in *Patterns in Java, Volume 3*) uses a proxy to enforce a security policy on access to a service-providing object.

Façade. The Façade pattern uses a single object as a front end to a set of interrelated objects rather than as a front end to a single object.

Object Request Broker. The Object Request Broker pattern (described in *Patterns in Java, Volume 3*) uses a proxy to hide the fact that a service object is located on a different machine than the client objects that want to use it.

Virtual Proxy. The Virtual Proxy pattern uses a proxy to create the illusion that a service-providing object exists before it has actually been created. The Virtual Proxy pattern is useful if the object is expensive to create and its services may not be needed. The copy-on-write proxy discussed under the Code Example heading for the Proxy pattern is a kind of virtual proxy.

Decorator. The Decorator pattern is structurally similar to the Proxy pattern in that it forces access to a service-providing object to be done indirectly through another object. The difference is a matter of intent. Instead of trying to manage the service, the indirection object in some way enhances the service.