6 Using design patterns

In the previous chapter, we looked at UML class diagrams. This chapter continues the study of the static view of software by looking at typical patterns found in class diagrams. These patterns recur in many designs; by learning and using them you are reusing the collective experience of many software developers.

In this chapter you will learn about the following

- The structure and format used to describe patterns.
- Patterns used to help ensure separation of concerns in a class diagram: Abstraction–Occurrence, Observer, and Player–Role.
- A pattern used to create hierarchies of instances: General Hierarchy.
- The Delegation pattern in which a method simply calls another method in another class, but which can significantly improve the overall design of a system.
- Patterns in which you use delegation to gain access to facilities in one or more other classes: Adapter, Façade and Proxy.
- Patterns that help protect other objects from unanticipated access: Immutable and Read-Only Interface.
- A pattern that enables you to create application-specific objects in a framework: Factory.

6.1 Introduction to patterns

As you gain experience in object-oriented software development, you will begin to notice that many parts of your models or designs reappear, with only slight changes, in many different systems or subsystems. These recurring aspects are called *patterns*. Many of them have been systematically documented for all software developers to use.

Definition: a *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.

In this chapter we will restrict our attention to patterns used in modeling and design; but people have also developed patterns for many other human tasks, such as doing business, diagnosing mechanical problems or taking better photographs. In Chapter 9, we will look at architectural patterns, which are used at the very highest level of software design.

A good pattern should be as general as possible, containing a solution that has been proven to solve the problem effectively in the indicated context. The pattern must be described in an easy-to-understand form so that people can determine when and how to use it. Studying patterns is an effective way to learn from the experience of others.

Each pattern should have a name; it should also have the following information:

- **Context:** the general situation in which the pattern applies.
- **Problem:** a sentence or two explaining the main difficulty being tackled.
- **Forces:** the issues or concerns that you need to consider when solving the problem. These include criteria for evaluating a good solution.
- **Solution:** the recommended way to solve the problem in the given context. The solution is said to 'balance the forces'; in other words, it has a good combination of advantages, with few counterbalancing disadvantages.
- **Antipatterns:** (optional) solutions that are inferior or do not work in this context. The reason for their rejection should be explained. The antipatterns may be valid solutions in other contexts, or may never be valid. They often are mistakes made by beginners.
- **Related patterns:** (optional) patterns that are similar to this pattern. They may represent variations, extensions or special cases.
- **References:** acknowledgements of those who developed or inspired the pattern.

A pattern should normally be illustrated using a simple diagram and should be written using a narrative writing style.

In the following sections, we present a sample of the most useful patterns that you can apply when you perform modeling and design using UML class diagrams. You will recognize some of them from the discussions in the last chapter. Here, however, we will describe them in a more formal way. Our list of patterns is by no means exhaustive – you should see the references at the end of the chapter to learn about the wide variety of patterns that are available.

The first three of the patterns described below we call *modeling patterns*, since they appear in domain models long before any software design occurs. The rest

Patterns and the patterns community

The word 'pattern' has a well-understood meaning in ordinary English: it refers to a set of instructions, or a model or an example from which things are made or matched.

The architect Christopher Alexander developed the notion of patterns for use in architecture and design. The software engineering community has adopted his meaning of the term.

Alexander defines patterns as 'a three-part rule, which expresses a relation between a certain context, a problem and a solution'.

Patterns are an excellent way to document design understanding, and to pass that understanding on to others who are learning how to design. By thinking about relevant patterns as they design, designers can work more rapidly and produce higher-quality work because they are reusing the experience of others.

The exact format you use to write patterns varies from author to author, but most authors include the same kind of information that we include. A group of interrelated patterns form a *pattern language*.

There is a 'patterns community' within software engineering. This is a loose but large collection of people who believe in the usefulness of patterns, as well as in certain philosophies regarding their development. Among the important philosophies of the patterns community are:

- Patterns represent well-known knowledge. In other words, no pattern could ever be patented because it must be in common use (note the use of a completely unrelated word 'patent' that sounds similar; for more on patents, see the sidebar in the risks section of Chapter 11). People do not *invent* patterns – a pattern is a literary work describing common practice.
- Patterns should be in the public domain, where possible; people should be encouraged to improve the pattern to make it more usable.
- When patterns are written or modified, they should be reviewed in a public setting by a group of the author's peers.
- Patterns should be written for the public good. A pattern that describes how to do something unethical, for example, would not be acceptable to the community.

of the patterns are *design patterns* since they involve details that would normally be deferred until the design stage.

6.2 The Abstraction–Occurrence pattern

Context This modeling pattern is most frequently found in class diagrams that form part of a system domain model.

Often in a domain model you find a set of related objects that we will call *occurrences*; the members of such a set share common information but also differ from each other in important ways.

Examples of sets of occurrences include:

- All the episodes of a television series. They have the same producer and the same series title, but different story-lines.

- The flights that leave at the same time every day for the same destination. They have the same flight number, but occur on different dates and carry different passengers and crew.
- All the copies of the same book in a library. They have the same title and author. However, the copies have different barcode identifiers and are borrowed by different people at different times.

Problem What is the best way to represent such sets of occurrences in a class diagram?

Forces You want to represent the members of each set of occurrences without duplicating the common information. Duplication would consume unnecessary space and would require changing all the occurrences when the common information changes. Furthermore, you want to avoid the risk of inconsistency that would result from changing the common information in some objects but not in others. Finally you want a solution that maximizes the flexibility of the system.

Solution Create an *«Abstraction»* class that contains the data that is common to all the members of a set of occurrences. Then create an *«Occurrence»* class representing the occurrences of this abstraction. Connect these classes with a one-to-many association. This is illustrated in Figure 6.1.

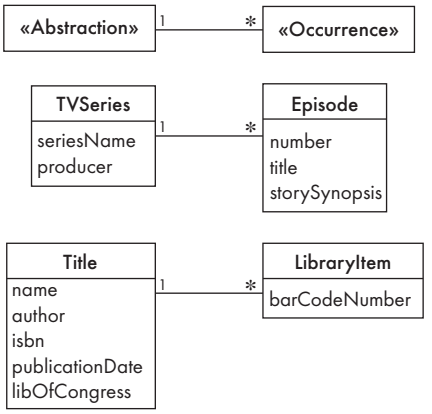

Figure 6.1 **Template and examples of the Abstraction–Occurrence pattern**

Examples You might create an «Abstraction» class called `TVSeries`; an instance of this might be the children's series 'Sesame Street'. You would then create an «Occurrence» class called `Episode`. Similarly, you might create an «Abstraction» class called `Title` which will contain the author and name of a book or similar publication. The corresponding «Occurrence» class might be called `LibraryItem`. These examples are illustrated in Figure 6.1.

Another example of the Abstraction–Occurrence design pattern is the pair consisting of `RegularFlight` and `SpecificFlight` in the airline system, as shown in Figure 5.29.

Antipatterns These antipatterns are examples of real mistakes made by beginners.

One inappropriate solution, shown in Figure 6.2(a), is to use a single class. This would not work, because information would have to be duplicated in each of the multiple copies of a book. For example, the different copies of *Moby Dick* can be borrowed by different people; there would, therefore, have to be separate 'Moby Dick' instances, listing the same name, author and ISBN.

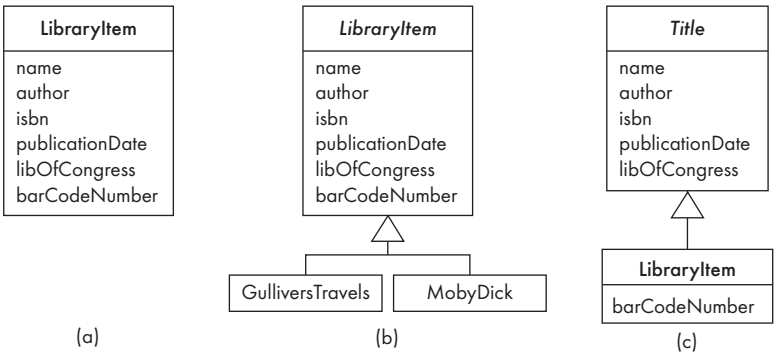

Figure 6.2 **Inappropriate ways to represent abstractions and occurrences**

Another very bad solution, shown in Figure 6.2(b), is to create a separate subclass for each title (i.e. a class `GulliversTravels` and another class `MobyDick`). Information such as name, author, etc. would still be duplicated in each of the instances of each subclass. Furthermore, this approach seriously restricts the flexibility of the system – you want to be able to add new books without having to program new classes.

Figure 6.2(c) shows another invalid solution: making the abstraction class a *superclass* of the occurrence class. This would not solve the original problem presented in this pattern, since, although the attributes in the superclass are inherited by the subclasses, the *data* in those attributes is not. For example, even though there is a `name` attribute defined in the superclass `Title`, we would still have to set the value of this attribute in every instance of `LibraryItem`.

Related patterns When the abstraction is an aggregate, the occurrences are also typically aggregates. The result is the Abstraction–Occurrence Square pattern, illustrated in Figure 6.3.

References This pattern is a generalization of the Title–Item pattern of Eriksson and Penker (see the 'For more information' section at the end of the chapter).

Exercise

E113 Apply the Abstraction–Occurrence pattern in the following situations. For each situation, show the two linked classes, the association between the classes, and the attributes in each class.

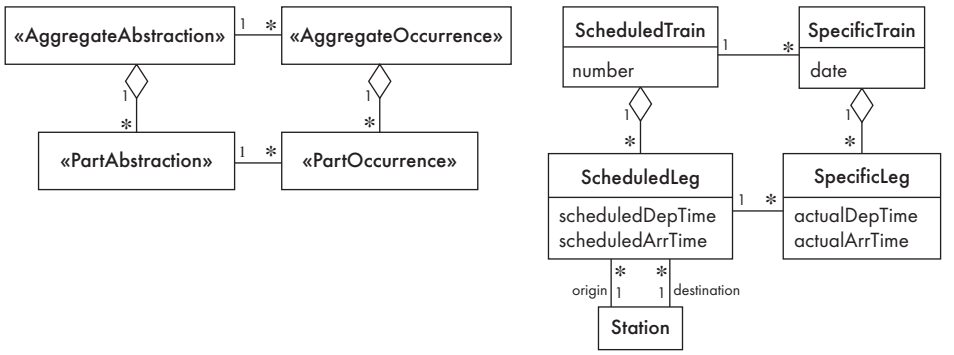

Figure 6.3 **The Abstraction–Occurrence Square pattern**

(a) The issues of a periodical.

(b) The copies of the issues of a periodical.

(c) The repeats and re-runs of the same television program.

(d) Models of electronic appliances and the individual appliances.

6.3 The General Hierarchy pattern

Context This modeling pattern occurs in many class diagrams. You often find a set of objects that have a naturally hierarchical relationship. For example, the relationships between managers and their subordinates in an organization chart, or the directories (also known as folders), subdirectories and files in a file system.

Each object in such a hierarchy can have zero or more objects above them in the hierarchy (superiors), and zero or more objects below them (subordinates). Some objects, however, cannot have any subordinates – for example the staff members in an organization chart (as opposed to the managers) or the files in a file system (as opposed to the directories).

Problem How do you draw a class diagram to represent a hierarchy of objects, in which some objects cannot have subordinates?

Forces You want a flexible way of representing the hierarchy that naturally prevents certain objects from having subordinates. You also have to account for the fact that all the objects share common features.

Solution Create an abstract «*Node*» class to represent the features possessed by each object in the hierarchy – one such feature is that each node can have superiors.

Then create at least two subclasses of the «Node» class. One of the subclasses, «*SuperiorNode*», must be linked by a «*subordinates*» association to the superclass; whereas at least one subclass, «*NonSuperiorNode*», must not be. The subordinates of a «SuperiorNode» can thus be instances of either «SuperiorNode» or «NonSuperiorNode».

The multiplicity of the «subordinates» association can be optional-to-many or many-to-many. If it is many-to-many, then the hierarchy of instances becomes a lattice, in which a node can have many superiors. The 'optional' allows for the case of the top node in the hierarchy, which has no superiors.

Examples In Figure 6.4, there are three kinds of employee in an organization, but only managers can supervise subordinates. Similarly, only directories can contain other file system objects. In a user interface, some objects (such as panels) can contain others; this is illustrated in Figure 6.5. As a final example, in Figure 5.30 a company can have customers, who in turn have other customers, and are the customers of other companies – the result is a network of customer/service-provider relationships.

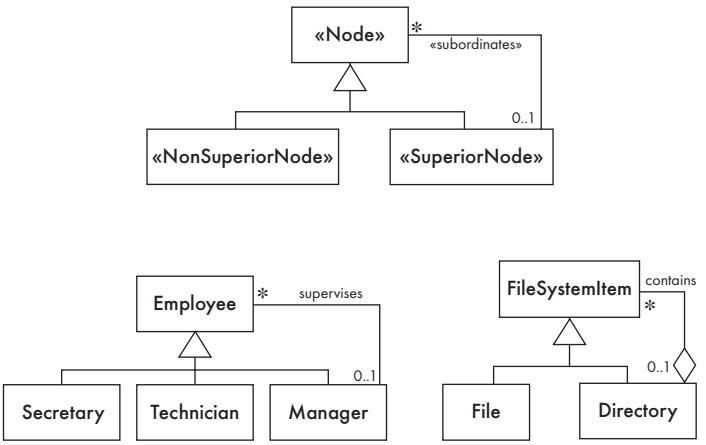

Figure 6.4 **Template and examples of the General Hierarchy design pattern**

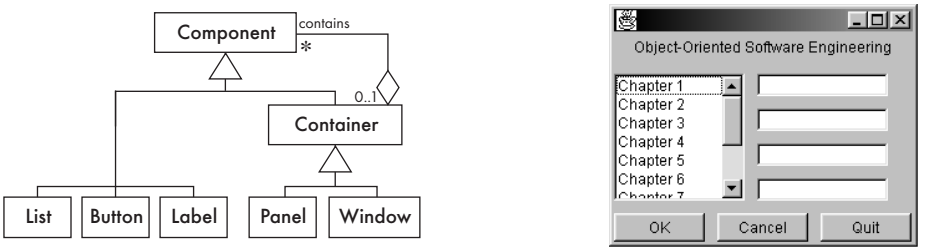

Figure 6.5 **Component classes whose instances form a hierarchy in a user interface. The example shows a frame, composed of a label (top), a list (left) and two panels. The first panel (on the right) is in turn composed of four text fields. The other panel (at the bottom) contains three buttons**

Antipatterns A common beginner's mistake is to model a hierarchy of categories using a hierarchy of classes, as illustrated in Figure 5.11. This mistake is made because beginners are taught about inheritance hierarchies, and then immediately think that every hierarchy should be an inheritance hierarchy.

Related patterns The Reflexive Association, discussed in Section 5.3, can be considered to be a pattern. Figures 5.8, 5.12(a) and 5.25(a) show that hierarchies of objects can be modeled using asymmetric reflexive associations. Doing so, however, does not allow for the special «NonSuperiorNode» classes that occur in the context of the General Hierarchy pattern.

The *Composite* pattern is a specialization of the General Hierarchy pattern. In the Composite pattern, the association between «SuperiorNode» and «Node» is an aggregation. A Composite is a recursive container; that is, a container that can contain other containers. This is the case with directories in a file system (Figure 6.4) as well as instances of `GUIComposite` (Figure 6.5).

References The Composite pattern is one of the 'Gang of Four' patterns. See the book by Gamma, Helm, Johnson and Vlissides in 'For more information'.

Exercises

E114 Figure 5.21 shows a hierarchy of vehicle parts. Show how this hierarchy might be better represented using the General Hierarchy pattern (or more precisely, by the Composite pattern).

E115 Revisit Exercise E91. Did you use the General Hierarchy pattern? If not, then redo the exercise, showing a hierarchy of various levels of government. Imagine that municipalities are the lowest levels of government.

E116 An expression in Java can be broken down into a hierarchy of subexpressions. For example, `(a/(b+c))+(b-func(d)*(e/(f+g)))` has `(a/b+c))` as one of its higher-level subexpressions.

(a) Using the General Hierarchy or Composite pattern, create a class diagram to represent expressions in Java. Hints:

(i) A higher-level expression might have more than two parts.
(ii) The parts of a higher-level expression are connected by operators; how can these be represented?
(iii) Think about what the «NonSuperiorNodes» must be.
(iv) Some expressions are surrounded by parentheses, while others need not be; how can this be represented?

(b) Using your class diagram from part (a), create an object diagram for the example above.

6.4 The Player–Role pattern

Context This modeling pattern can solve modeling problems when you are drawing many different types of class diagram. A *role* is a particular set of features

associated with an object in a particular context. An object may *play* different roles in different contexts.

For example, a student in a university can be either an undergraduate student or a graduate student at any point in time – and is likely to need to change from one of these roles to another. Similarly, a student can also be registered in his or her program full-time or part-time, as shown in Figure 5.16; in this case, a student may change roles several times. Finally, an animal may play several of the roles shown in Figure 5.15, although in this case the roles are unlikely to change.

Problem How do you best model players and roles so that a player can change roles or possess multiple roles?

Forces It is desirable to improve encapsulation by capturing the information associated with each separate role in a class. However, as discussed in Chapter 5, you want to avoid multiple inheritance. Also, you cannot allow an instance to change class.

Solution Create a «*Player*» class to represent the object that plays different roles. Create an association from this class to an abstract «*Role*» class, which is the superclass of a set of possible roles. The subclasses of this «Role» class encapsulate all the features associated with the different roles.

If the «Player» can only play one role at a time, the multiplicity between «Player» and «Role» can be one-to-one, otherwise it will be one-to-many.

Instead of being an abstract class, the «Role» can be an interface. The only drawback to this variation is that the «Role» usually contains a mechanism, inherited by all its subclasses, allowing them to access information about the «Player». Therefore you should only make «Role» an interface if this mechanism is not needed.

Examples Figure 6.6 shows how an `Animal` or a `Student` can take on several roles to solve the problems posed by Figures 5.14 and 5.15.

The example in the middle of Figure 6.6 shows that an object can have a varying number of roles. An animal could be aquatic, land-based or both. Note that we could also have used a role class to record whether an animal is a carnivore, herbivore or omnivore; however, since this information remains the same for the life of an animal, we can just use ordinary subclasses.

The bottom example in Figure 6.6 shows that you can have two separate «Role» superclasses. In this case, a student is characterized by his or her attendance status (full-time or part-time) and by whether he or she is a graduate or undergraduate. Both of these types of status can change during the life of a `Student` object. The Player–Role pattern with two one-to-one associations makes it possible to create a full-time undergraduate student, a part-time graduate student or any other combination.

The Player–Role pattern is also used in the airline system to allow a person to be both a passenger and an employee. This is illustrated in Figure 5.31.

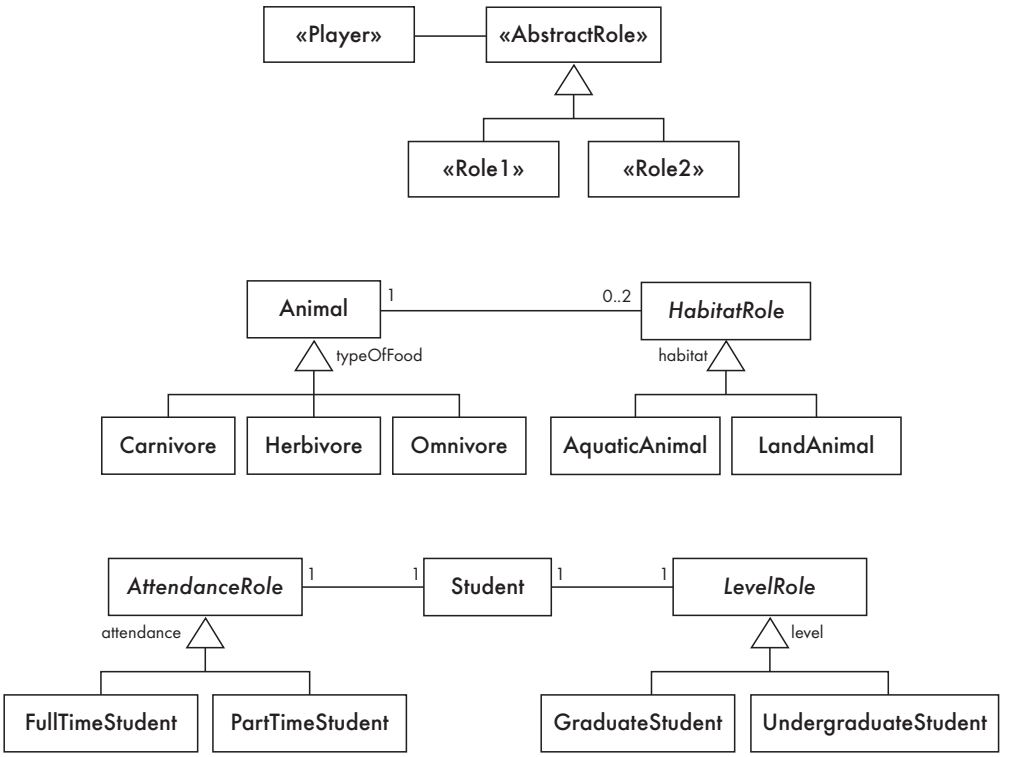

Figure 6.6 Template and examples of the Player–Role design pattern

Antipatterns One way to implement roles is simply to merge all the features into a single «Player» class and not have «Role» classes at all. This, however, creates an overly complex class – and much of the power of object orientation is lost.

You could also create roles as subclasses of the «Player» class. But, as we have already discussed, this is a bad idea if it results in the need for multiple inheritance or requires an instance to change class.

Related patterns The Abstraction–Occurrence pattern has a similar structure to the Player–Role pattern: the player has many roles associated with it, just like the abstraction has many occurrences. However, the semantics of the two patterns is quite different. A key distinction is that in the Abstraction–Occurrence pattern, an abstraction is, as the name says, abstract, while its occurrences tend to be real-world things such as copies of books. The inverse is true in the Player–Role pattern: the player is normally a real-world entity (e.g. a person) while its roles are abstractions.

References This pattern appears in the OMT book by Rumbaugh et al. (1991), referred to in the 'For more information' section of Chapter 5. At the time that book was written, however, the term 'pattern' had not yet taken on its current use.

Exercise

E117 Draw a class diagram, applying the Player–Role pattern in the following circumstances.

(a) Users of a system have different privileges.

(b) Managers can be given one or more responsibilities.

(c) Your favorite sport (football, baseball, etc.) in which players can play at different positions at different times in a game or in different games.

6.5 The Singleton pattern

Context In software systems, it is very common to find classes for which you want only one instance to exist. Such a class is called a *singleton*.

For example, the `Company` or `University` classes in systems that run the business of that company or university might be singletons. Another example is the `MainWindow` class in a graphical user interface for systems that can only have one main window open.

Problem How do you ensure that it is never possible to create more than one instance of a singleton class?

Forces If you use a public constructor, you cannot offer the guarantee that no more than one instance will be created.

The singleton instance must also be accessible to all classes that require it, therefore it must often be public.

Solution In a singleton class, create the following:

- A private class variable, often called `theInstance`. This stores the single instance.
- A public class method (static method), often called `getInstance`. The first time this method is called, it creates the single instance and stores it in `theInstance`. Subsequent calls simply return `theInstance`.
- A private constructor, which ensures that no other class will be able to create an instance of the singleton class.

Example In an employee management system (Figure 6.7), the `Company` class might be the central class that encapsulates several important features related to the system as a whole. The Singleton implementation ensures that only one instance of this important class can exist. The public class method `getInstance` makes this instance globally accessible.

Note The Singleton pattern should not be overused, since the singleton instance is effectively a global variable, and the use of global variables should be minimized.

References The Singleton pattern is one of the 'Gang of Four' patterns.

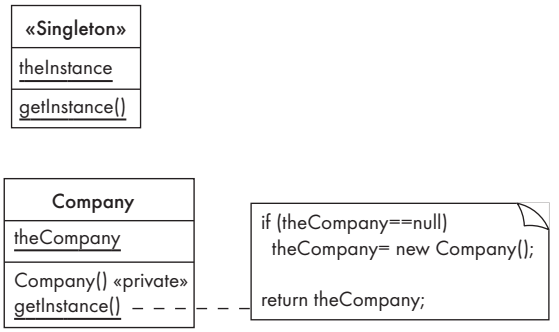

Figure 6.7 **Template and example of the Singleton design pattern**

Exercise

E118 Discuss how the Singleton pattern could be generalized to the case where a class could be limited to have a maximum of N instances.

6.6 The Observer pattern

Context When you create a two-way association between two classes, the code for the classes becomes inseparable. When you compile one, the other one has to be available since the first one explicitly refers to it. This means that if you want to reuse one of the classes, you also have to reuse the other; similarly, if you change one, you probably have to change the other.

Problem How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems? In other words, how do you ensure that an object can communicate with other objects without knowing which class they belong to?

Forces You want to maximize the flexibility of the system to the greatest extent possible.

Solution Create an abstract class we will call the *«Observable»* that maintains a collection of «*Observer*» instances. The «Observable» class is very simple; it merely has a mechanism to add and remove observers, as well as a method `notifyObservers` that sends an `update` message to each «Observer». Any application class can declare itself to be a subclass of the «Observable» class. This is illustrated in Figure 6.8.

«Observer» is an interface, defining only an abstract `update` method. Any class can thus be made to observe an «Observable» by declaring that it implements the interface, and by asking to be a member of the observer list of the «Observable». The «Observer» can then expect a call to its `update` method whenever the «Observable» changes.

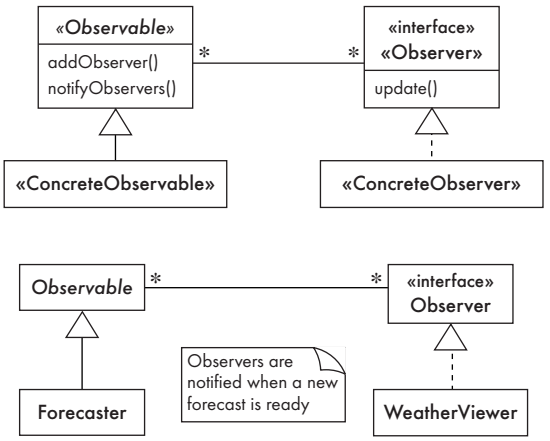

Figure 6.8 Template and example of the Observer design pattern

Using this pattern, the «Observable» neither has to know the nature or the number of the classes that will be interested in receiving the `update` messages, nor what they will do with this information.

Examples Java has an `Observer` interface and an `Observable` class. The Java mechanism is a specific implementation of this pattern.

In order to obtain a weather forecast, a system has to perform a long sequence of calculations. Suppose that these computations are under the control of a `Forecaster` object, as shown in Figure 6.8. When a new forecast is ready this object notifies all interested instances. The `Forecaster` is therefore the observable object. One observer might be a user interface object responsible for displaying the weather forecast. Another observer might use weather forecasts to plan the schedule of some workers – the workers might do one set of tasks on rainy days and another set on sunny days.

The Observer pattern is widely used to structure software cleanly into relatively independent modules. It is the basis of the MVC architecture presented in Chapter 9. It is also used in an additional layer of the OCSF framework as presented in Section 6.14.

Antipatterns Beginners tend to connect an observer directly to an observable so that they both have references to each other. We pointed out earlier that this means you cannot plug in a different observer.

Another mistake is to make the observers subclasses of the observable. This will not work because then each observer is at the same time an observable. It is not therefore possible to have more than one observer for an observable.

References The Observer pattern is one of the 'Gang of Four' patterns. It is also widely known as Publish-and-Subscribe (the observers are *Subscribers* and the observable is a *Publisher*).

Exercises

E119 Look at the Java documentation and explain the similarities or differences between the mechanism behind the `ActionEvent` and `ActionListener` classes and the Observer pattern.

E120 Use the Observer pattern to model a small system where several different classes would be notified each time an item is added or removed from an inventory.

6.7 The Delegation pattern

Context You need an operation in a class and you realize that another class already has an implementation of the operation. However, it is not appropriate to make your class a subclass and inherit this operation, either because the isa rule does not apply, or because you do not want to reuse *all* the methods of the other class.

Problem How can you most effectively make use of a method that already exists in the other class?

Forces You want to minimize development cost and complexity by reusing methods. You want to reduce the linkages between classes. You want to ensure that work is done in the most appropriate class.

Solution Create a method in the «Delegator» class that does only one thing: it calls a method in a neighboring «Delegate» class, thus allowing reuse of the method for which the «Delegate» has responsibility. By 'neighboring', we mean that the «Delegate» is connected to the «Delegator» by an association. This is illustrated in Figure 6.9.

Normally, in order to use delegation an association should already exist between the «Delegator» and the «Delegate». This association may be bidirectional or else unidirectional from «Delegator» to «Delegate». However, it may sometimes be appropriate to create a new association just so that you can use delegation – provided this does not increase the overall complexity of the system.

Delegation can be seen as providing selective inheritance.

Examples As shown in Figure 6.9, a `Stack` class can be easily created from an existing collection class such as `LinkedList`. The `push` method of `Stack` would simply call the `addFirst` method of `LinkedList`, the `pop` method would call the `removeFirst` method and the `isEmpty` method would delegate to the method of the same name. The other methods of the `LinkedList` class would not be used since they do not make sense in a `Stack`.

The bottom example in Figure 6.9 shows two levels of delegation in the airline system. `Booking` has a `flightNumber` method that does nothing other than delegate to the method of the same name in its neighbor, `SpecificFlight`. This in turn delegates to `RegularFlight`.

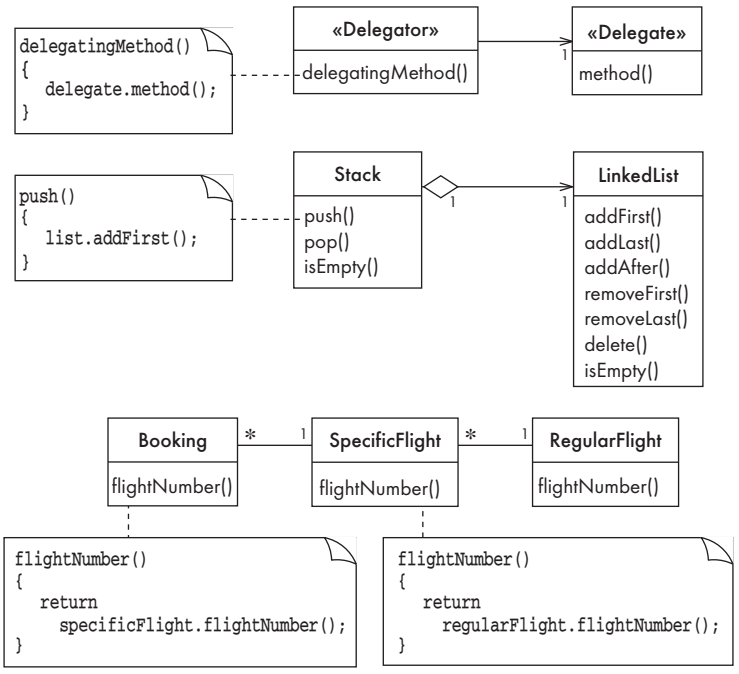

Figure 6.9 **Template and examples of the Delegation design pattern**

Antipatterns The Delegation pattern brings together three principles that encourage flexible design:

(a) favoring association instead of inheritance when the full power of inheritance is not needed;

(b) avoiding duplication of chunks of code; and

(c) accessing 'nearby' information only. Violation of any of these principles should be avoided, as explained below.

Instead of using delegation, it is common for people to overuse generalization and *inherit* the method that is to be reused – for example, making `Stack` a subclass of `LinkedList`. The biggest problem with this is that some of the methods of `LinkedList`, such as `addAfter`, do not make sense in a `Stack`, yet they would be available.

Instead of creating a single method in the «Delegator» that does nothing other than call a method in the «Delegate», you might consider having many different methods in the «Delegator» call the delegate's method. For example, all the methods in `Booking` could independently call `specificFlight.flightNumber()`. Unfortunately, this would create many more linkages in the system. It would be better to ensure that only *one* method in the «Delegator» calls the method in the «Delegate». Otherwise, when you make a change to the method in the «Delegate», you may have to change all of the calling methods.

The Law of Demeter

A fundamental principle of the Delegation pattern is that a method should only communicate with objects that are neighbors in the class diagram. This is a special case of the 'Law of Demeter', which was formulated by a team from Northeastern University in Boston.

The Law of Demeter says, in short, 'only talk to your immediate friends'. In software design, this means that a method should only access data passed as arguments, linked via associations, or obtained via calls to operations on other neighboring data. The rationale is that this limits the impact of changes, and makes it easier for software designers to understand that impact. If each method only communicates with its neighbors, then it should only be impacted when those neighbors change, not when changes occur in more distant parts of the system.

The Law of Demeter was named after Demeter, the ancient Greek goddess of agriculture, because its developers were interested in 'growing' software incrementally. Adhering to the Law of Demeter should make incremental development much easier.

Finally, you want to ensure that in delegation a method only accesses neighboring classes. For example it would not be good for `Booking`'s `flightNumber` method to be written as:

```
return specificFlight.regularFlight.flightNumber();
```

This is bad because the further a method has to reach to get its data, the more sensitive it becomes to changes in the system. Maintenance becomes easier if you know that a change to a class will only affect its neighbors.

Related patterns The Adapter and Proxy patterns, discussed below, both use delegation.

References The Delegation pattern is mentioned in the book by Grand (see 'For more information' at the end of the chapter).

Exercise

E121 Find as many situations as you can where the Delegation pattern should be applied in Figure 5.25(c).

6.8 The Adapter pattern

Context You are building an inheritance hierarchy and you want to incorporate into it a class written by somebody else – that is, you want to reuse an existing unrelated class. Typically the methods of the reused class do not have the same name or argument types as the methods in the hierarchy you are creating. The reused class is also often already part of its own inheritance hierarchy.

Problem How do you obtain the power of polymorphism when reusing a class whose methods have the same function but do not have the same signature as the other methods in the hierarchy?

Forces You do not have access to multiple inheritance or you do not want to use it.

Solution Rather than directly incorporating the reused class into your inheritance hierarchy, instead incorporate an «*Adapter*» class, as shown in Figure 6.10. The «Adapter» is connected by an association to the reused class, which we will call the «*Adaptee*». The polymorphic methods of the «Adapter» *delegate* to methods of the «Adaptee». The delegate method in the «Adaptee» may or may not have the same name as the delegating polymorphic method.

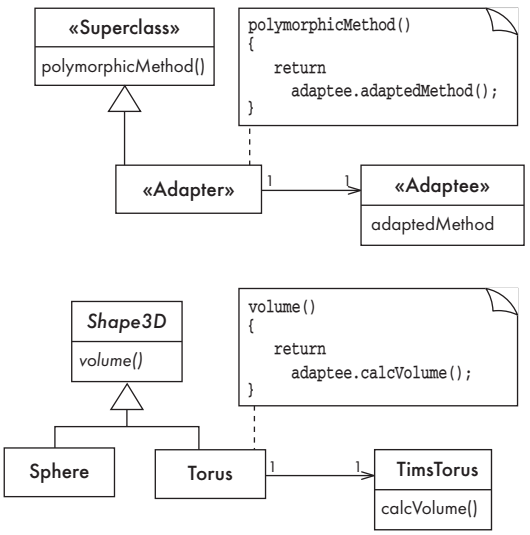

Figure 6.10 **Template and example of the Adapter design pattern**

Other code that accesses facilities of an «Adapter» object will be unaware that it is indirectly using the facilities of an instance of the «Adaptee».

Instead of being part of an inheritance hierarchy, an «Adapter» can be one of several classes that implement an interface.

Example As shown in Figure 6.10, imagine you are creating a hierarchy of three-dimensional shapes. However, you want to reuse the implementation of an equivalent class called `TimsTorus`. You do not want to modify the code of `TimsTorus`, since it is also being used by others; therefore you cannot make `TimsTorus` a subclass of `Shape3D`. You therefore make `Torus` an «Adapter». Its instances have a link to an instance of `TimsTorus`, and delegate all operations to `TimsTorus`.

Adapters are sometimes called *wrappers*. The Java wrapper classes `Integer`, `Float`, `Double` etc. are adapters for the Java primitive types.

A variation of the Adapter design pattern is used in an extended version of the OCSF framework, as explained in Section 6.14.

Related patterns The Adapter pattern is one of several patterns that make it easier to use other classes. Other patterns discussed below that do this are:

- **Façade:** provides a single class to make it easy to access a whole subsystem of classes.
- **Read-Only Interface:** provides an interface that prevents changing instances of another class.
- **Proxy:** provides a lightweight class that makes it unnecessary to always have to deal with a heavyweight class.

References The Adapter pattern is one of the 'Gang of Four' patterns.

Exercise

E122 Explain whether or not the `MouseAdapter` class defined in Java is an implementation of the Adapter pattern.

6.9 The Façade pattern

Context Often, an application contains several complex packages. A programmer working with such packages has to manipulate many different classes.

Problem How do you simplify the view that programmers have of a complex package?

Forces It is hard for a programmer to understand and use an entire subsystem – in particular, to determine which methods are public. If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

Solution Create a special class, called a «Façade», which will simplify the use of the package. The «Façade» will contain a simplified set of public methods such that most other subsystems do not need to access the other classes in the package. The net result is that the package as a whole is easier to use and has a reduced number of dependencies with other packages. Any change made to the package should only necessitate a redesign of the «Façade» class, not classes in other packages.

Example The airline system discussed in Chapter 5 has many classes and methods. Other subsystems that need to interact with the airline system risk being 'exposed' to any changes that might be made to it. We can therefore define the class `Airline` to be a «Façade», as shown in Figure 6.11. This provides access to the most important query and booking operations.

References This pattern is one of the 'Gang of Four' patterns.

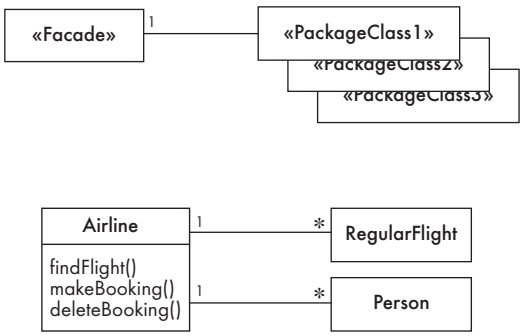

Figure 6.11 **Template and example of the Façade design pattern**

Exercise

E123 Suppose that you want to be able to use different database systems in different versions of an application. To facilitate interchanging databases, you create a Façade interface plus Façade classes associated with each specific database system. Draw the class diagram that corresponds to this.

6.10 The Immutable pattern

Context An immutable object is an object that has a state that never changes after creation. An important reason for using an immutable object is that other objects can trust its content not to change unexpectedly.

Problem How do you create a class whose instances are immutable?

Forces The immutability must be enforced. There must be no loopholes that would allow 'illegal' modification of an immutable object.

Solution Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified. In other words, make sure that any instance methods have no *side effects* by changing instance variables or calling methods that themselves have side effects. If a method that would otherwise modify an instance variable *must* be present, then it has to return a new instance of the class.

Example In an immutable `Point` class, the `x` and `y` values would be set by the constructor and never modified thereafter. If a `translate` operation were allowed to be performed on such a `Point`, a new instance would have to be created. The object that requests the `translate` operation would then make use of the new translated point.

In Figure 2.8, imagine there was a method `changeScale(x,y)` in an immutable version of class `Circle`. This would return the same object if `x` and `y` were both 1.0, a new `Circle` if `x` and `y` were both equal, and a new `Ellipse` otherwise. Similarly, the `changeScale(x,y)` method in an immutable `Ellipse` class would

return a new `Circle` if the `changeScale(x,y)` would result in the semi-major axis equaling the semi-minor axis.

Related patterns The Read-Only Interface pattern, described next, provides the same capability as Immutable, except that certain privileged classes are allowed to make changes to instances.

References This pattern was introduced by Grand (see 'For more information' at the end of the chapter).

Exercise

E124 Imagine that all the classes in Figure 2.8 were immutable. What other methods might be added to the system that would return instances of a *different* class from the class in which they are written?

6.11 The Read-Only Interface pattern

Context This is closely related to the Immutable pattern. You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable.

Problem How do you create a situation where some classes see a class as read-only (i.e. the class is immutable) whereas others are able to make modifications?

Forces Programming languages such as Java allow you to control access by using the `public`, `protected` and `private` keywords. However, making access public makes it public for both reading and writing.

Solution Create a «*Mutable*» class as you would create any other class. You pass instances of this class to methods that need to make changes.

Then create a public interface we will call the «*ReadOnlyInterface*», that has only the read-only operations of «Mutable» – that is, only operations that *get* its values. You pass instances of the «ReadOnlyInterface» to methods that do not need to make changes, thus safeguarding these objects from unexpected changes. The «Mutable» class implements the «ReadOnlyInterface».

This solution is shown in Figure 6.12.

Example Figure 6.12 shows a `Person` interface that can be used by various parts of the system that have no right to actually modify the data. The `MutablePerson` class exists in a package that protects it from unauthorized modification.

The Read-Only Interface design pattern can also be used to send data to objects in a graphical user interface. The read-only interface ensures that no unauthorized modifications will be made to this data. We present this usage in the description of the MVC architecture in Chapter 9.

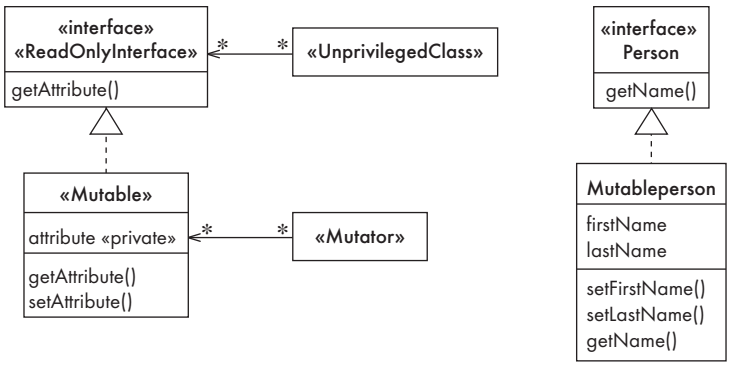

Figure 6.12 **Template and example of the Read-Only Interface design pattern**

Antipattern You could consider making the read-only class a subclass of the «Mutable» class. But this would not work since whole point of this pattern is that you want a single class with different sets of access rights.

References This pattern was introduced by Grand (see 'For more information' at the end of the chapter).

Exercise

E125 Create a read-only version of the `SpecificFlight` class shown in Figures 5.32 and 5.33. The purpose of this would be so that you could pass instances to other subsystems safely.

6.12 The Proxy pattern

Context This pattern is found in class diagrams that show how aspects of the architecture of a system will be implemented.

Often, it is time-consuming and complicated to obtain access in a program to instances of a class. We call such classes *heavyweight* classes. Instances of a heavyweight class might, for example, always reside in a database. In order to use the instances in a program, a constructor must load them with data from the database. Similarly, a heavyweight object may exist only on a server: before using the object, a client has to request that it be sent; the client then has to wait for the object to arrive.

In both the above situations, there is a time delay and a complex mechanism involved in creating the object in memory. Nevertheless, many other objects in the system may want to refer to or use instances of heavyweight classes.

It is very common for all the domain classes to be heavyweight classes. Sets of instances of these must also be managed by heavyweight versions of the collection classes used to implement associations (such as `ArrayList` or `Vector`).

Problem How can you reduce the need to load into memory large numbers of heavyweight objects from a database or server, when not all of them will be needed?

A related problem is this: if you load one object from a database or server, how can you avoid loading all the other objects that are linked to it?

Forces You want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities. It is also important for many objects to persist from run to run of the same program.

However, in a large system it would be impractical for all the objects to be loaded into memory whenever a program starts. Memory size is limited; it takes a long time to load a database into memory; and only a small number of the objects in the database will actually be needed during any particular run of a program. Keeping all objects in memory would also make it difficult for multiple programs to share the same objects.

It would be ideal to be able to program the application *as if* all the objects were located in memory. The details of how the objects are actually stored and loaded should be *transparent* to the programmer. This provides for separation of concerns: some programmers can worry about loading and saving of objects, while others can be concerned with implementing the responsibilities of the domain model.

Solution Create a simpler version of the «HeavyWeight» class. We will call this simpler version a «Proxy». The «Proxy» has the *same interface* as the «HeavyWeight», therefore programmers can declare variables without caring whether a «Proxy» or its «HeavyWeight» version will be put in the variable. This is illustrated in Figure 6.13.

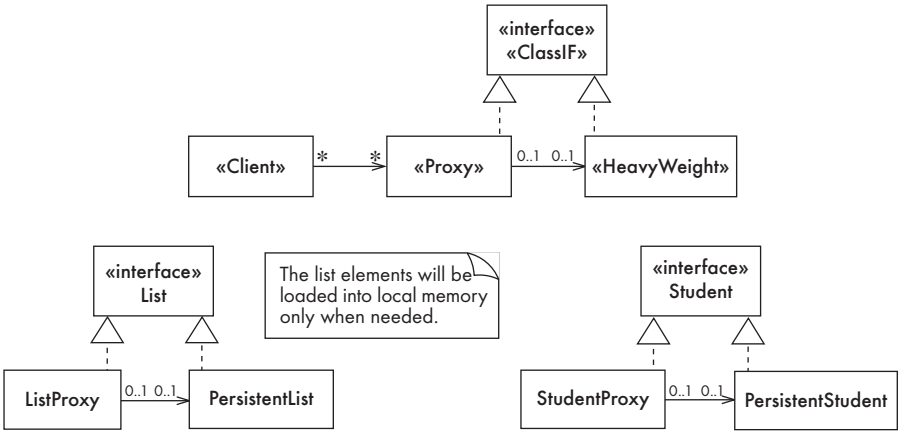

Figure 6.13 **Template and examples of the Proxy design pattern**

The «Proxy» object normally acts only as a placeholder. If any attempt is made to perform an operation on the proxy then the proxy *delegates* the operation to the «HeavyWeight». When needed, the «Proxy» undertakes the expensive task

of obtaining the real «HeavyWeight» object. The proxy only needs to obtain the «HeavyWeight» once; subsequently it is available in memory and access is therefore fast.

Some proxies may have implementations of a limited number of operations that can be performed without the effort of loading the «HeavyWeight».

In some systems, most of the variables manipulated by the domain model actually contain instances of «Proxy» classes.

Examples In Figure 6.13, a software designer may declare that a variable is to contain a `List`. This variable would, however, actually contain a `ListProxy`, since it would be expensive to load an entire list of objects into memory, and the list might not actually be needed. However, as soon as an operation accesses the list, the `ListProxy` might at that point create an instance of `PersistentList` in memory. On the other hand, the `ListProxy` might be able to answer certain queries, such as the number of elements in the list, without going to the effort of loading the `PersistentList`.

Now, imagine that the `PersistentList` was actually a list of students. These objects might *also* be proxies – in this case, instances of `StudentProxy`. Again, instances of `PersistentStudent` would only be loaded when necessary.

The Proxy pattern is widely used in many software architectures. We will discuss it again in the context of the Broker architectural pattern in Chapter 9.

Antipatterns Instead of using proxy objects, beginner designers often scatter complex code around their application to load objects from databases.

A strategy that only works for very small systems is to load the whole database into memory when the program starts.

Related patterns The Proxy pattern is one of several patterns that obtain their power from delegating responsibilities to other classes, hence it uses the Delegation pattern.

References The Proxy pattern is one of the 'Gang of Four' patterns.

Exercise

E126 Discuss the advantages of using an image-proxy when manipulating the photos in a digital photo album application. What operations could conceivably be performed by the proxy without loading the heavyweight image into memory?

6.13 The Factory pattern

Context You have a reusable framework that needs to create objects as part of its work. However, the class of the created objects will depend on the application.

Problem How do you enable a programmer to add a new application-specific class «ApplSpecificClass» into a system built on such a framework? And how do you

arrange for the framework to instantiate this class, without modifying the framework?

Forces You want the benefits of a framework, but retain the flexibility of having the framework create and work with application-specific classes that the framework does not yet know about.

Solution The framework delegates the creation of instances of «ApplSpecificClass» to a specialized class «ApplSpecificFactory». The «ApplSpecificFactory» implements a generic interface «Factory» defined in the framework. The «Factory» declares a method whose purpose is to create some subclass «AppSpecificClass» of a class we will call «GenericClass». This is illustrated in Figure 6.14.

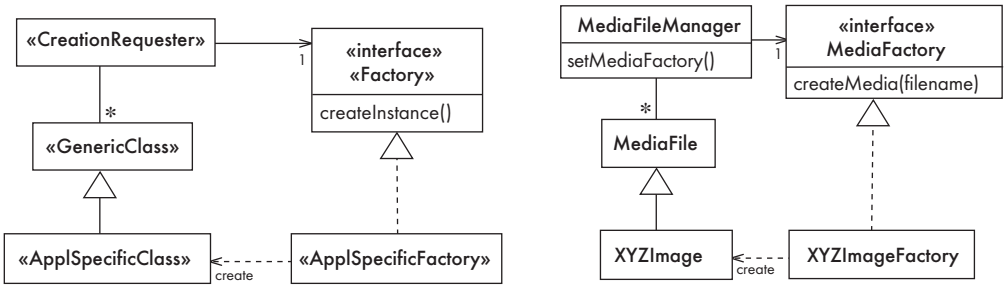

Figure 6.14 **Template and example of the Factory design pattern**

Example As is shown in right-hand diagram of Figure 6.14, suppose you have a framework with a class called `MediaFile`, whose subclasses will represent the contents of various types of audio-visual media files. You want to create a subclass of this, `XYZImage`, to represent your new media format; and you need to ensure the framework class `MediaFileManager` can instantiate this. You therefore create a factory class, `XYZImageFactory`, whose sole job is to create instances of `XYZImage`. This implements the `MediaFactory` interface. You initiate the system to create `XYZImage`s by calling the `setMediaFactory` method of `MediaFileManager`; this creates a link to an instance of `XYZImageFactory`. Each time the `MediaFileManager` needs to create a new `MediaFile` object, it calls the `createMedia` method as implemented by `XYZImageFactory`; this then creates a new `XYZImage`.

Antipatterns You could get rid of the factory and instead modify the framework code in `MediaFileManager` to force it to always instantiate `XYZImage` directly. However, this would only be possible if you have access to the source of the framework; and would not work if you needed to be able to work with several different «Factory»–«ApplSpecificClass» pairs. As a general principle, modifying a framework should always be considered forbidden.

References The Factory pattern is one of the 'Gang of Four' patterns.

Exercises

E127 In a given game, carnivore and herbivore animals are created at random instants by the game engine. Depending on the country selected by the user, a factory for the appropriate carnivores and herbivores is loaded (e.g. one that will create lions and gazelles in Kenya, but cougars and beavers in Canada). Draw the class diagram to represent this idea.

E128 Find the design pattern that would be most appropriate for the following problems:

(a) You are building an inheritance hierarchy of products that your company sells; however, you want to reuse several classes from one of your suppliers. You cannot modify your suppliers' classes. How do you ensure that the facilities of the suppliers' classes can still be used polymorphically?

(b) You want to allow operations on instances of `RegularPolygon` that will distort them such that they are no longer regular polygons. How do you allow the operations without raising exceptions?

(c) Your program manipulates images that take a lot of space in memory. How can you design your program so that images are only in memory when needed, and otherwise can only be found in files?

(d) You have created a subsystem with 25 classes. You know that most other subsystems will only access about 5 methods in this subsystem; how can you simplify the view that the other subsystems have of your subsystem?

(e) You are developing a stock quote framework. Some applications using this framework will want stock quotes to be displayed on a screen when they become available; other applications will want new quotes to trigger certain financial calculations; yet other applications might want both of the above, plus having quotes transmitted wirelessly to a network of pagers. How can you design the framework so that various different pieces of application code can react in their own way to the arrival of new quotes?

E129 The `Iterator` interface, as defined in Java, is an implementation of what is called the 'Iterator' design pattern. Study the Java documentation describing `Iterator`, then using the format discussed in this chapter, write a description of the Iterator pattern, with sections that define its context, problem, forces and solution.

E130 (Advanced) In order to improve the access to information stored in a database, several applications use the concept of a cache. The basic principle is to keep in local memory objects that would normally be destroyed, because it is expected that these objects will be requested again later on. In this way, when they are indeed required again, access to them is very fast.

(a) Create a design pattern that describes this idea. Use the format presented in this chapter.

(b) Scan the literature on design patterns and look for the Cache Management design pattern. Compare it with the solution you proposed.

6.14 Enhancing OCSF to employ additional design patterns

The Object Client–Server Framework (OCSF) presented in Chapter 3 provides a simple way to set up a client–server application rapidly. In this section, we introduce additional features of OCSF and show how the use of design patterns can greatly increase flexibility. As with the basic classes of OCSF, code for the extensions discussed here is available on the book's web site (http://www.lloseng.com).

Client connection factory

The first extension to the basic framework is the addition of a *Factory* to handle client connections. To understand the usefulness of this mechanism, let us first review client connection management on the server side. Each time a new client connects to the server, a `ConnectionToClient` object is created. This object defines a thread that manages all communication with that particular client. All messages received from the client are passed on to the `handleMessageFromClient` method in a subclass of `AbstractServer`. This method is *synchronized* so that if two `ConnectionToClient` threads need to access the same resource (e.g. an instance variable of the server) then they won't interfere with each other – only one call to `handleMessageFromClient` will execute at a time.

However, there are some circumstances when you might want to allow developers to create application-specific subclasses of `ConnectionToClient`:

- You might not like having all message handling processed sequentially in the synchronized `handleMessageFromClient` in the server object. Instead you might want to have client message handling take place in a version of `handleMessageFromClient` in a special subclass of `ConnectionToClient`. This could still be synchronized if you like, but it would be synchronized on the `ConnectionToClient` object in order that the processing of messages from different clients could be done concurrently.
- You might want to have different `handleMessageFromClient` methods in different subclasses of `ConnectionToClient`. A different subclass of `ConnectionToClient` could, for example, be created to handle clients in your local area network, as opposed to clients somewhere else on the Internet.

To enable the server class to instantiate an application-specific subclass of `ConnectionToClient`, OCSF provides an optional Factory mechanism. There are two keys to this. The first key is an interface called `AbstractConnectionFactory` (see Figure 6.15). You create an application-specific factory class that implements the

`createConnection` method in this interface. Your factory class will in turn create instances of your own subclass of `ConnectionToClient`. The second key is the method `setConnectionFactory` found in `AbstractServer`. Your server class calls this to ensure that whenever a new client attempts to connect, your factory will be directed to instantiate your subclass of `ConnectionToClient` to handle the connection.

To use the OCSF factory mechanism, you therefore need to do the following:

1. Create your subclass of `ConnectionToClient`. Its constructor must have the same signature as `ConnectionToClient`, and it must call the constructor of `ConnnectionToClient` using the `super` keyword. Your class will also normally want to override `handleMessageFromClient`; if this method returns `true`, the version of `handleMessageFromClient` in your server class will *also* be subsequently called.

2. Create your factory class that simply defines a method for the `createConnection` operation of the `AbstractConnectionFactory` interface. Typically, the method would look like this:

```
protected ConnectionToClient createConnection(
   ThreadGroup group, Socket clientSocket,
   AbstractServer server) throws IOException
{
   return new Connection(group,clientSocket,server);
}
```

3. Arrange for the server make the following call before it starts listening:

```
setConnectionFactory(new MyConnectionFactory());
```

Observable layer

A second extension to the OCSF framework is the addition of an Observable layer. We will describe the client side, but the server side works the same way.

In the basic `OCSF`, a message received by a client is processed by the subclass of `AbstractClient` that implements the `handleMessageFromServer` abstract method. Each time a new application is developed, therefore, the `AbstractClient` class must be subclassed.

The Observer pattern provides an alternative mechanism for developing a client. Any number of «Observer» classes can ask to be notified when something 'interesting' happens to the client – the arrival of a message or the closing of a connection, for example. We would therefore like to have a subclass of `AbstractClient` that is an «Observable». Unfortunately, since Java does not permit multiple inheritance, we cannot make it a subclass of the `Observable` class itself. Instead, we use the Adapter pattern, as shown in Figure 6.15.

The extended OCSF has the class `ObservableClient`. This has exactly the same interface as `AbstractClient`, except that it is a subclass of `Observable`. It is also an adapter: it delegates methods such as `sendToServer`, `setPort`, etc. to instances of a concrete subclass of `AbstractClient` called `AdaptableClient`. Designers using `ObservableClient` never need to know that `AdaptableClient` exists.

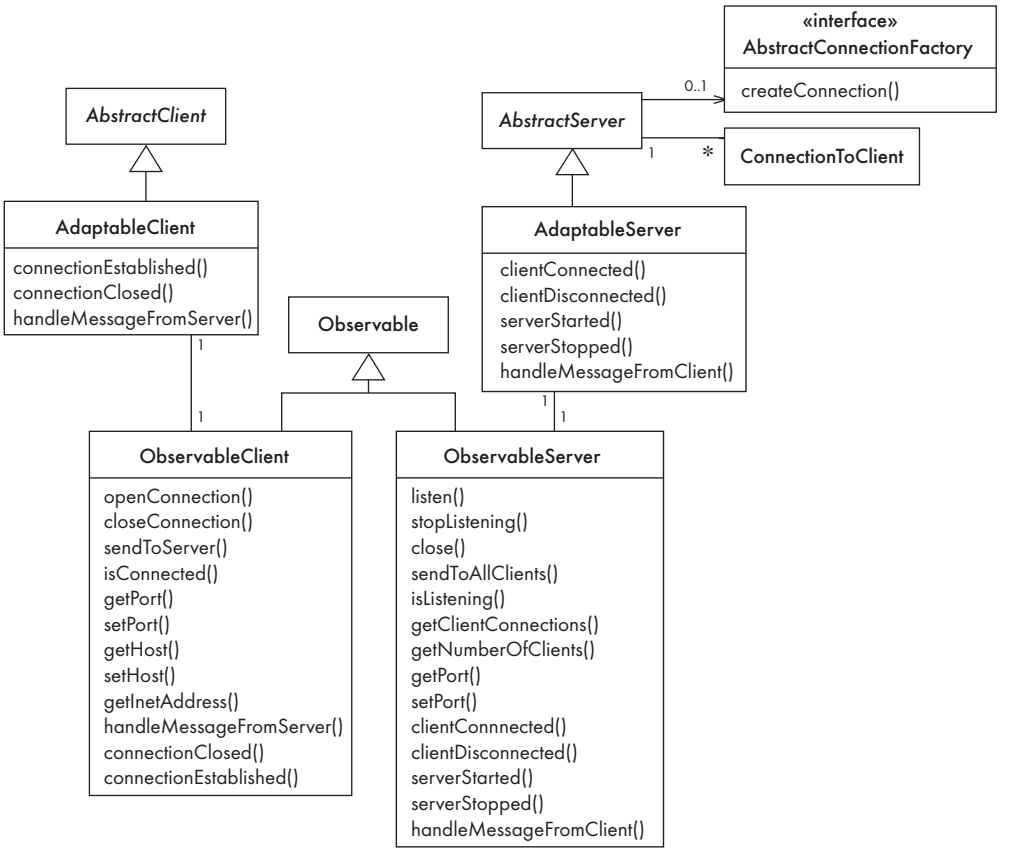

Figure 6.15 **The Object Client–Server Framework with extensions to employ the Observable and Factory design patterns**

Implementation of the Observable layer

The following are some of the highlights of the implementation of the client side:

- The class `AdaptableClient`, as the concrete subclass of `AbstractClient`, provides the required concrete implementation of `handleMessageFromServer`. It also provides implementations of the hook methods `connnectionClosed` and `connectionEstablished`. All that these three callback methods do is delegate to the `ObservableClient`. Their structure is as follows:

```
callbackMethod()
{
  observable.callbackMethod();
}
```

- There is always a one-to-one relationship between an `AdaptableClient` and an `ObservableClient`. Instances of both these classes must exist.

- All the service methods in `ObservableClient` (such as `openConnection`) simply delegate to the `AdaptableClient`. They have the following structure:

```
serviceMethod()
{
  return adaptable.serviceMethod();
}
```

- The method `handleMessageFromServer` in `ObservableClient` is implemented as follows:

```
public void handleMessageFromServer(Object message)
{
  setChanged();
  notifyObservers(message);
}
```

- The other callback methods in `ObservableClient`, such as the hook method `connectionClosed`, do nothing. A designer could elect to create a subclass of `ObservableClient` which might implement `connectionClosed` like this:

```
public void connectionClosed()
{
  setChanged();
  notifyObservers("connectionClosed");
}
```

The server side is implemented analogously, except that the instance of `ConnectionToClient` could also be sent to the observers.

Some important advantages of using the Observable layer of OCSF are:

1. Different types of messages can be processed by different classes of observer. For example, different parts of a user interface might update themselves when specific messages are received; they would ignore the other messages.
2. Programmers using the `ObservableClient` or `ObservableServer` need to know very little about these classes. There is thus a better separation of concerns between the communication subsystem (OCSF) and different application subsystems.

Exercise

E131 In the Observable layer of OCSF, the classes `ObservableClient` and `ObservableServer` are similar to adapters in the sense that their main function is to delegate to the adaptable classes. In what way do they differ from true adapters? You can look at the design presented above to answer this, but it may also help if you study the source code.

Using the Observable layer

In order to connect a class to the observable layer of OCSF, the procedure is as follows:

1. Create the application class that implements the `Observer` interface (note that this is not a subclass of any of the framework classes).
2. Register an instance of the new class as an observer of the `ObservableClient` (or `ObservableServer`). Typically, you would do this in the constructor, in the following manner:

```
public MessageHandler(Observable client)
{
  client.addObserver(this);
  ...
}
```

3. Define the `update` method in the new class. Normally a given class will react only to messages of a particular type. In the following example, our application class is only interested in messages that are of class `SomeClass`.

```
public void update(Observable obs, Object message)
{
  if (message instanceOf SomeClass)
  {
    // process the message
  }
}
```

If `message` is a `String`, a condition in the `if` block could be added to determine what to do with the message.

6.15 Difficulties and risks when using design patterns

The following are the key difficulties to anticipate when designing and using design patterns:

- **Patterns are not a panacea**. Whenever you see an indication that a pattern should be applied, you might be tempted to apply the pattern blindly. However, this can lead to unwise design decisions. For example, you do not always need to apply the Façade pattern in every subsystem; adding the extra class might make the overall design more complex, especially if instances of many of the classes in the subsystem are passed as data to methods outside the subsystem.
  Resolution. Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces. Also, make sure you justify each design decision carefully.
- **Developing patterns is hard**. Writing a good pattern takes considerable work. A poor pattern can be hard for other people to apply correctly, and can lead

them to make incorrect decisions. It is particularly hard to write a set of forces effectively.

Resolution. Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns. Take an in-depth course on patterns. Iteratively refine your patterns, and have them peer reviewed at each iteration.

6.16 Summary

Applying patterns to the process of creating class diagrams helps you to create better models. Patterns help you to avoid common mistakes and to create systems that are simpler and more flexible.

Some of the more important patterns that occur in domain models include Abstraction–Occurrence, General Hierarchy and Player–Role. Observer, Adapter and Factory are patterns that frequently occur in complete system class diagrams. Immutable, Façade and Proxy are typically applied when the modeler is moving towards a more detailed stage of design.

The patterns can also be categorized according to the principles they embody. The Delegation pattern is a fundamental pattern that prevents excessive interconnection among different parts of a system. Abstraction–Occurrence, Observer and Player–Role also help increase separation of concerns. Adapter, Façade and Proxy help the developer to reuse the facilities of other classes. Immutable and Read-Only Interface help protect objects from unexpected changes.

6.17 For more information

The following are some of the many available resources about patterns:

- The Patterns Home Page: http://www.hillside.net/patterns/ – an extensive list of resources about patterns
- A reference source for design patterns in Java: http://www.fluffycat.com/java/patterns.html
- Brad Appleton's description of patterns: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html
- E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, October 1994. This book is the most widely cited book about patterns. Its authors are often referred to as the 'Gang of Four'
- C. Alexander, *A Pattern Language*, Oxford University Press, 1977. The classic book by the originator of the patterns movement

- H-E. Eriksson and M. Penker, *Business Modeling with UML: Business Patterns at Work*, Wiley, 2000
- M. Grand, *Patterns in Java Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML*, 2nd edition, Wiley, 2002
- W. H. Brown, R. C. Malveau, H. W. McCormick III and T. J. Mowbray, *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*, Wiley, 1998. A summary of many of the key mistakes that software developers and managers make

Project exercises

E132 Summarize the advantages and disadvantages of:

(a) The observable layer of OCSF.

(b) Creating a subclass of `ConnectionToClient` (and instantiating it using the factory).

E133 Draw an object diagram showing an instance of a subclass of `AbstractServer` with a `ConnectionFactory` that has created three instances of two different subclasses of `ConnectionToClient`

E134 Modify the SimpleChat system so that it uses the Observable layer of the OCSF. The number of changes you make should be minimized, and the external interface to the system should not change. When you complete this exercise, you will have completed Phase 4 of SimpleChat.

E135 Examine your class diagram for the Small Hotel Reservation System from Chapter 5.

(a) Determine which patterns, if any, you already applied, without knowing it.

(b) See if you can improve your class diagram by applying one or more of the patterns discussed in this chapter.