Object-Oriented Software Engineering

Practical Software Development using UML and Java

Second edition

Timothy C. Lethbridge
Robert Laganière

The McGraw-Hill Companies

London • Burr Ridge, IL • New York • St. Louis • San Francisco •Auckland
Bogotá • Caracas • Lisbon • Madrid • Mexico • Milan • Montreal • New Delhi
Panama • Paris• San Juan • São Paulo • Singapore •Tokyo • Toronto

9.2 Principles leading to good design

In this section we introduce you to general principles you should apply whenever you are designing software. Applying these principles diligently will result in designs that have many advantages over designs in which the principles were not applied.

Some overall goals we want to achieve when doing good design are:

- Increasing profit by reducing cost and increasing revenue. For most organizations, this is the central objective. However, there are a number of ways to reduce cost, and also many different ways to increase the revenue generated by software.
- Ensuring that we actually conform to the requirements, thus solving the customers' problems.
- Accelerating development. This helps reduce short-term costs, helps ensure the software reaches the market soon enough to compete effectively, and may be essential to meet some deadline faced by the customer.
- Increasing qualities such as usability, efficiency, reliability, maintainability and reusability. These can help reduce costs and also increase revenues.

Design Principle 1: Divide and conquer

The divide and conquer principle dates back to the earliest days of organized human activity. Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things. Military campaigns are waged this way: commanders try to avoid fighting on all fronts at once. Cars are also built using the divide and conquer strategy: some people design the engines while others design the body, etc. Furthermore, the task of assembling the car is also divided into smaller, more manageable chunks – each assembly-line worker will focus on one small task.

In software engineering, the divide and conquer principle is applied in many ways. We have already seen how the process of development is divided into activities such as requirements gathering, design and testing. In this section we will look at how software systems themselves can be divided.

Dividing a software system into pieces has many advantages:

- Separate people can work on each part. The original development work can therefore be done in parallel.
- An individual software engineer can specialize in his or her component, becoming expert at it. It is possible for someone to know everything about a small part of a system, but it is not possible to know everything about an entire system.
- Each individual component is smaller, and therefore easier to understand.

- When one part needs to be replaced or changed, this can hopefully be done without having to replace or extensively change other parts.
- Opportunities arise for making the components reusable.

A software system can be divided in many ways:

- A distributed system is divided up into clients and servers.
- A system is divided up into subsystems.
- A subsystem can be divided up into one or more packages.
- A package is composed of classes.
- A class is composed of methods.

Exercise

E171 In Exercises E55 to E57 and E61, you were asked to create requirements for four systems. Divide each of these into separate subsystems.

Design Principle 2: Increase cohesion where possible

The cohesion principle is an extension of the divide and conquer principle – divide and conquer simply says to divide things up into smaller chunks. Cohesion says to do it intelligently: yes, divide things up, but keep things together that belong together.

A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things. This makes the system as a whole easier to understand and change.

Listed below are several important types of cohesion that designers should try to achieve. Table 9.1 summarizes these types of cohesion, starting with the most desirable.

Functional cohesion This is achieved when a module only performs a single computation, and returns a result, without having side effects.

A module lacks side effects if performing the computation leaves the system in the same state it was in before performing the computation. The result computed by the module is the only thing that should have an effect on subsequent computations.

The inputs to a functionally cohesive module typically include function parameters, but they can also include files or some other stream of data. Whenever exactly the same inputs are provided, the module will always compute the same result. The result is often a simple return value, but can also be a more complex data structure.

Modules that update a database or create a new file are not functionally cohesive since they have side effects in the database or file-system respectively. Similarly, a module that interacts with the user is not functionally cohesive:

Table 9.1 **The different types of cohesion, ordered from highest to lowest in terms of the precedence you should normally give them when making design decisions**

Cohesion type	*Comments*
Functional	Facilities are kept together that perform only *one computation* with no *side effects*. Everything else is kept out
Layer	*Related services* are kept together, everything else is kept out, and there is a *strict hierarchy* in which higher-level services can access only lower-level services. Accessing a service may result in side effects
Communicational	Facilities for operating on the *same data* are kept together, and everything else is kept out. Good classes exhibit communicational cohesion
Sequential	A set of procedures, which work in sequence to perform some computation, is kept together. *Output from one is input to the next*. Everything else is kept out
Procedural	A set of procedures, which are called *one after another*, is kept together. Everything else is kept out
Temporal	Procedures used in the *same general phase* of execution, such as initialization or termination, are kept together. Everything else is kept out
Utility	*Related utilities* are kept together, when there is no way to group them using a stronger form of cohesion

prompting the user is a kind of output, therefore it violates the rule that the only output of a functionally cohesive module is the result returned at the end of execution.

The following are some examples of modules that can be designed to be functionally cohesive:

- A module that computes a mathematical function such as sine or cosine.
- A module that takes a set of equations and solves for the unknowns.
- A module in a chemical factory that takes data from various monitoring devices and computes the yield of a chemical process as a percentage of the theoretical maximum.

A functionally cohesive module can call the services of other modules, but the called modules must preserve the functional cohesion. For example, a module that computes a mathematical function can certainly call modules that perform other mathematical functions.

There are several reasons why it is good to achieve functional cohesion:

- It is easier to understand a module when you know that all it does is generate one specific output and has no side effects.
- Due to its lack of side effects, a functionally cohesive module is much more likely to be reusable.
- It is easier to replace a functionally cohesive module with another that performs the same computation. Being able to make such easy replacements greatly assists maintenance. In the case of a non-functionally cohesive module that has side effects, you would have to verify that any replacement also has precisely the same side effects. Even if the side effects were carefully documented, doing such verification is time-consuming and error-prone. Furthermore, maintainers often fail to pay attention to the presence of side effects.

Layer cohesion This is achieved when the facilities for providing a set of related *services* to the user or to higher-level layers are kept together, and everything else is kept out.

To have proper layer cohesion, the layers must form a hierarchy. Higher layers can access services of lower layers, but it is essential that the lower layers do not access higher layers. This is illustrated in Figure 9.3.

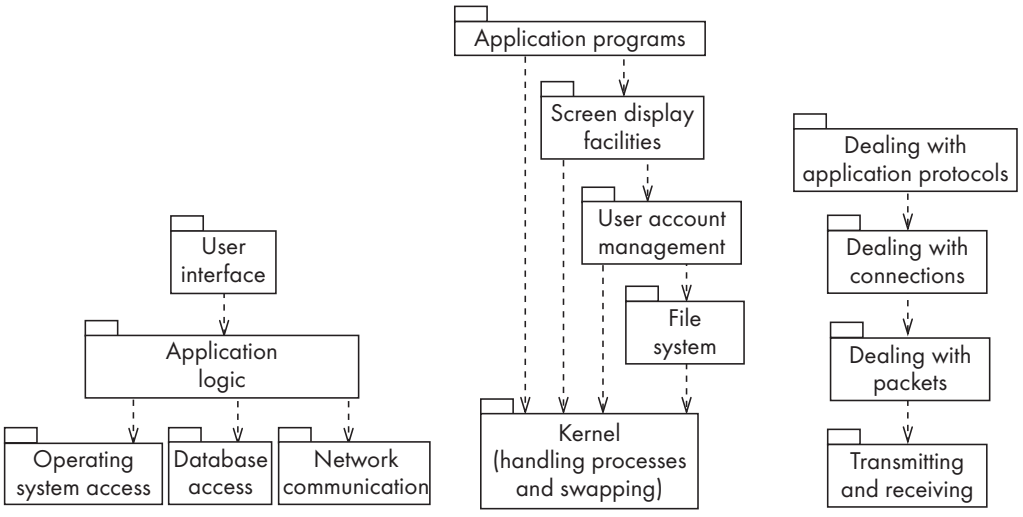

Figure 9.3 **Examples of the use of layers. Higher layers can call on the services of lower layers, but not the other way around**

An individual service in a layer may have functional cohesion. However, this is not necessary – side effects are allowed, and are often essential.

The set of related services that could form a layer might include:

- services for computation;

- services for transmission of messages or data;
- services for storage of data;
- services for managing security;
- services for interacting with users;
- services to access the operating system;
- services to interact with the hardware.

For example, if a system is to interface with a particular sound card, then a module should be created specifically to interact with that card. Furthermore, all the code for directly accessing the card should be in this module, and the module should do nothing else except interact with the card.

The set of procedures or methods through which a layer provides its services is commonly called an *application programming interface* (API). The specification of the API must describe the protocol that higher-level layers use to access it, as well as the semantics of each service, including the side effects.

Advantages of layer cohesion are:

- You can replace one or more of the top-level layers without having any impact on the lower-level layers.
- You know you can replace a lower layer with an equivalent layer, because you know it does not access higher layers. To do this, however, you have to replicate all aspects of the API, so that upper layers will continue to work the same way.

We will revisit the notion of layers when we discuss architectural patterns, in Section 9.6.

Communicational cohesion

This is achieved when modules that access or manipulate certain data are kept together (e.g. in the same class) – and everything else is kept out. One of the strong points of the object-oriented paradigm is that it helps ensure communicational cohesion – provided the principles of object orientation discussed in Chapters 2, 5 and 6 are properly followed.

The term 'communicational' is used for historical reasons. You can remember it by thinking of the following: All the procedures that 'communicate' with the data are kept together.

For example, a class called `Employee` would have good communicational cohesion if all the system's facilities for storing and manipulating employee data were contained in this class, and if the class did not do anything other than manage employee data.

As another example of communicational cohesion, imagine a module that updates a database, and a second module that keeps a history log of the changes to the database. Since both database and log file are representations of the same data, both modules should be kept together in a higher-level module or subsystem.

A communicationally cohesive module can be embedded in a layer. In other words, part of a layer's API can involve manipulating a particular class of data. The objects manipulated by the layer may be returned to higher layers in response to calls to the API.

The big advantage of communicational cohesion is the same key advantage we ascribed earlier in this book to object orientation: when you need to make changes to the data, you will find all the code in one place.

You should not sacrifice layer cohesion to achieve communicational cohesion: for example, even though objects may be stored in a database or on a remote host, a class must only load and save objects using the services in the API of lower layers.

Figure 9.4 shows several examples of communicationally cohesive modules (marked with a 'C'). These exist inside layers (marked 'L') and call on services in their own layer as well as lower layers. The services they call on may be in modules with other types of cohesion.

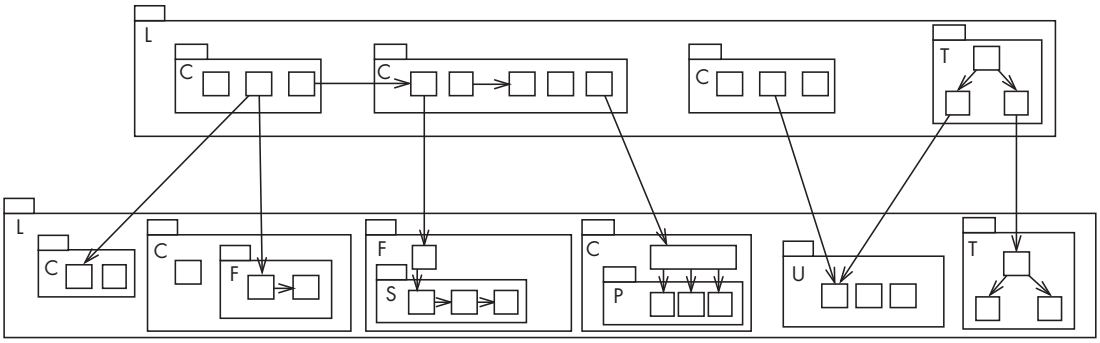

Figure 9.4 **Cohesive modules, nested inside each other, using the services of other modules. The modules are labeled using the first letter of the type of cohesion they represent**

Sequential cohesion This is achieved when a series of procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out. This is illustrated in Figure 9.4 by the module marked 'S'.

Your objective should be to achieve sequential cohesion, once you have already achieved the other types of cohesion listed above. Methods in two different classes might provide inputs to each other and be called in sequence; but they would each be kept in their own class, since communicational cohesion is more important than sequential cohesion.

As an example of sequential cohesion, imagine a text recognition subsystem. One module is given a bitmap as input and divides it up into areas that appear to contain separate characters. The output from this is fed into a second module that recognizes shapes and determines the probability that each area corresponds to a particular character. The output from that is fed into a third module that uses the probabilities to determine the sequence of words embedded in the input. If all these modules were grouped together, then the result would have sequential cohesion.

Procedural cohesion This is achieved when you keep together several procedures that are used one after another, even though one does not necessarily provide input to the next. It is therefore weaker than sequential cohesion. In Figure 9.4, the module marked 'P' is procedurally cohesive.

For example, in a university registration system, there would be a module to perform all the steps required to register a student in a course. The facilities for doing separate activities, such as adding a new course, would be in other modules.

Temporal cohesion This is achieved when operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out. This is weaker than procedural cohesion and is illustrated in Figure 9.4 by the modules marked 'T'.

For example, a designer would achieve temporal cohesion by placing together the code used during system start-up or initialization, so long as this did not violate one of the other forms of cohesion listed above. Similarly, all the code for system termination, or for certain occasionally used features, could be kept together to achieve temporal cohesion.

There may be a temporally cohesive module in a layer whose job is to initialize the services of that layer. The module would be called at startup time, and not at any other time.

Although it would be temporally cohesive, it would be a violation of communicational cohesion to create a module that *directly* initializes the static variables of several different classes or the services of different layers. However, it would be permissible to have a temporally cohesive module that *calls* the initialization procedures of other modules.

Utility cohesion This is achieved when related utilities that cannot be logically placed in other cohesive units are kept together. A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable. A utility module is marked 'U' in Figure 9.4.

For example, the `java.lang.Math` class has utility cohesion. Where possible, it would be better to put mathematical functions in classes on whose instances they are applied; however, `java.lang.Math` allows the grouping together of functions that have no obvious single home.

Exercises

E172 Categorize the following aspects of a design by the types of cohesion that they would exhibit if properly designed:

(a) All the information concerning bookings is kept inside a particular class, and everything else is kept out.

(b) A module is created to convert a bitmap image to the JPEG format.

(c) A separate subsystem is created that runs every night to generate statistics about the previous day's sales.

(d) A data processing operation involves receiving input from several sources, sorting it, summarizing information by input source, sorting according to the input source that generated the most data and then returning the results for the use of other subsystems. The code for these steps is all kept together, although utilities are called to do operations such as sorting.

E173 What is wrong with the following designs from the perspective of cohesion, and what could be done to improve them?

(a) There are two subsystems in a university registration system that do the following. *Subsystem A* displays lists of courses to a student, accepts requests from the student to register in courses, ensures that the student has no schedule conflicts and is eligible to register in the courses, stores the data in the database and periodically backs up the database. *Subsystem B* allows faculty members to input student grades, and allows administrators to assign courses to faculty members, add new courses, and change a student registration. It also prints the bills that are sent to students.

(b) In an electronic commerce application, a module is created to add books to the 'shopping basket' and perform such operations as computing the total amount the customer owes. A second module adds 'special reward' merchandise to the shopping basket; this module also displays the contents of the shopping basket on the screen and sends an email to the user telling him or her what he or she bought.

E174 Describe the kinds of cohesion present in the SimpleChat system.

Design Principle 3: Reduce coupling where possible

Coupling occurs when there are interdependencies between one module and another. Figure 9.5 illustrates the concept of a tightly coupled and loosely coupled system.

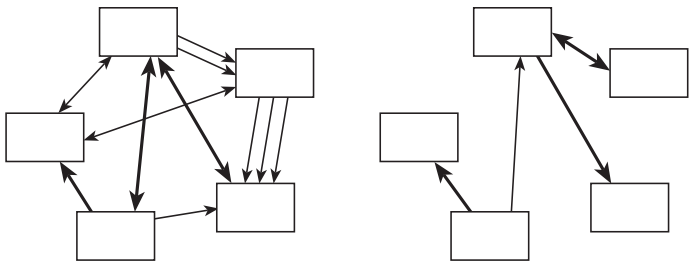

Figure 9.5 **Abstract examples of a tightly coupled system (left) and a loosely coupled system (right). The boldness of the arrows indicates the strength of the coupling**

In general, the more tightly coupled a set of modules is, the harder it is to understand and, hence, change the system. Two reasons for this are:

- When interdependencies exist, changes in one place will require changes somewhere else. Requiring changes to be made in more than one place is problematic since it is time-consuming to find the different places that need changing, and it is likely that errors will be made.
- A network of interdependencies makes it hard to see at a glance how some component works.

Additionally, coupling implies that if you want to reuse one module, you will also have to import those with which it is coupled. This is because the coupled components need each other in order to work properly.

Below, we list some of the many different ways by which modules can be coupled, and some of the ways to reduce the coupling. The types of coupling are summarized in Table 9.2.

To reduce coupling, you have to reduce the *number* of connections between modules and the *strength* of the connections. Some of the types of coupling listed in Table 9.2 are particularly strong and should always be avoided.

Content coupling This occurs when one component *surreptitiously* modifies data that is *internal* to another component. Content coupling should always be avoided since any modification of data should be easy to find and easy to understand.

Java is designed so that the worst kinds of content coupling (e.g. those involving manipulation of pointers) cannot be easily achieved. However, there are still some unfortunate tricks that Java programmers can play.

A form of content coupling occurs whenever you modify a public instance variable in a way that designers did not intend. To reduce content coupling you should therefore *encapsulate* all instance variables by declaring them `private`, and providing get and set methods. If you do this, you then have confidence that the only places where the variable is accessed and modified are in these methods. The set methods can ensure that only valid changes are made to the variables.

A worse form of content coupling, which is much harder to detect, occurs when you directly modify an instance variable *of* an instance variable. For example, in the following code, class `Arch` has a method called `slant`; this surreptitiously modifies the `y` value of the `Point` at the end of its `baseline` instance variable.

```
public class Line
{
  private Point start, end;
 ...
  public Point getStart() {return start;}
  public Point getEnd() {return end;}
}
```

Table 9.2 **Different types of coupling. You should reduce coupling where possible, but the types at the top are the strongest and hence the most important to avoid**

Coupling type	*Comments*
Content	A component *surreptitiously modifying internal data* of another component. Always avoid this
Common	The use of *global variables*. Severely restrict this
Control	One procedure *directly controlling another* using a flag. Reduce this using polymorphism
Stamp	One of the *argument types* of a method is one of your *application classes*. If it simplifies the system, replace each such argument with a simpler argument (an interface, a superclass or a few simple data items)
Data	The use of *method arguments that are simple data*. If possible, reduce the number of arguments
Routine call	*A routine calling another*. Reduce the total number of separate calls by encapsulating repeated sequences
Type use	The use of a *globally defined data type*. Use simpler types where possible (superclasses or interfaces)
Inclusion/ import	*Including a file or importing a package*. Eliminate when not necessary
External	A *dependency exists to elements outside the scope* of the system, such as the operating system, shared libraries or the hardware. Reduce the total number of places that have dependencies on such external elements

```
public class Arch
{
  private Line baseline;
 ...
  void slant(int newY)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(),newY);
  }
}
```

The content coupling occurs here even though the instance variables are private, and `baseline`, an instance of `Line`, is supposedly immutable (`Line` has no `setStart` or `setEnd` methods). It is surreptitious because the `Line` is changed without 'knowing' it is changing.

Part of the problem is that this code does not adhere to the delegation pattern (and the law of Demeter) as discussed in Chapter 6: the `slant` method is not accessing a neighboring object (the `Line`) but a more distant object (the `Point`).

Two things must be done to combat this form of content coupling:

1. Make moving the end of a `Line` *explicit*, by adding a `moveEnd` method to it. The `slant` method should call this. However, this is not enough since programmers could still bypass the `moveEnd` method.

2. Make the `Line` class truly immutable. To do this it is necessary to use immutable classes for its instance variables. If you do this, then you eliminate the possibility of surreptitious modification. You will notice that the `PointCP` class discussed in Chapter 2 was immutable.

Common coupling

This occurs whenever you use a global variable – all the modules using the global variable become coupled to each other, and to the module that declares the variable. The coupling occurs because changes to the variable's declaration will affect all the code that uses the variable. Also, changes to the way one module uses a variable will often have an effect on how the other modules should interpret the variable.

The word 'global', as used here, can mean that the variable is visible to all procedures and objects in the system. However, a weaker form of common coupling occurs any time a variable can be accessed by all instances of a subset of the system's classes (e.g. a Java package).

In older programming languages, the use of global variables was widespread; the name 'common' comes from the Fortran language in which it is the keyword used to declare global data. In Java, public static variables serve as global variables.

The use of common coupling should be minimized, since it shares many of the disadvantages of content coupling. Occasionally, a case can be made to create global variables that represent system-wide default values – the argument for this is that it would be more complex to force a large number of routines to pass around such information as their parameters. However, most of these system-wide values are actually constants (i.e. declared `final`), and not variables. For example, the `java.lang.Math` package has the constants `PI` and `E`.

As is the case with content coupling, common coupling can be reduced by encapsulation. For each global variable, create a module that has specially designated public methods that can be called to get or set the data. The internal representation of the data can then be more easily changed and it can be protected from inappropriate changes made by 'rogue' code; also, the set method can verify that changes are valid.

Encapsulation reduces the harm of global variables, but there is still some undesirable coupling, therefore avoid having too many such encapsulated variables. Note that the Singleton pattern, discussed in Chapter 6, provides encapsulated global access to an object; therefore avoid having too many singletons.

Control coupling This occurs when one procedure calls another using a 'flag' or 'command' that explicitly controls what the second procedure does. The following is an example:

```
public routineX(String command)
{
  if (command.equals("drawCircle")
  {
    drawCircle();
  }
  else
  {
    drawRectangle();
  }
}
```

The method `routineX` will have to change whenever any of its callers adds a new command. It should also probably be changed if any of its callers deletes a command, otherwise it will have code that is said to be 'dead'.

Control coupling can often be reduced by simply having the callers of `routineX` directly call methods such as `drawCircle` or `drawRectangle`. But the use of polymorphic operations is normally the best way to reduce control coupling. In the example above, there could be two separate classes `Circle` and `Rectangle`; `routineX` could then just call `draw`, with the system choosing the appropriate method to run.

There are cases when control coupling cannot or should not be completely avoided. For example, the SimpleChat server has the method `handleMessageFromClient`. This is tightly coupled to the methods in the SimpleChat client that generate the commands. One way to reduce the control coupling in this case would be to have a look-up table that mapped a command to a method that should be called when that command is issued. There is still some coupling, since the look-up table must be modified when commands are changed; however, look-up tables are simpler in structure than nested if-then-else statements.

Stamp coupling This occurs whenever one of your application classes is declared as the type of a method argument. Some stamp coupling is necessary; however, the following situation illustrates why it is best to try to reduce it.

Imagine a class `Employee` that has many instance variables such as `name`, `address`, `email`, `salary`, `manager`, etc., and many methods to manipulate these variables. Any method that is passed an instance of `Employee` is given the ability to call any of its public methods. The method `sendEmail` in the following `Emailer` class, for example, has this ability.

```
public class Emailer
{
  public void sendEmail(Employee e, String text) {...}
 ...
}
```

The problem here is that the `sendEmail` method *does not need* to be given access to the full `Employee` object; it really only needs access to `email` and `name`. Giving it full access represents unnecessary stamp coupling. Any time a maintainer changes the `Employee` class he or she will have to check the `sendEmail` method to see if it needs to be changed. The `Emailer` class is also not reusable – it can only be used in applications that use the `Employee` class.

There are two ways to reduce stamp coupling, a) using an interface as the argument type, and b) passing simple variables. The following illustrates the first way:

```
public interface Addressee
{
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee {...}

public class Emailer
{
  public void sendEmail(Addressee e, String text) {...}
 ...
}
```

The stamp coupling is reduced since the `sendEmail` method now has access only to the `name` and `email` data that it truly needs. Changes to the `Employee` class will be far less likely to impact it. The `sendEmail` method will still be impacted if the `Addressee` interface is changed, although that is probably unlikely to occur. Given that the `Addressee` interface is easy to reuse, the `Emailer` class now becomes reusable.

Instead of creating a new `Addressee` interface, you might have considered using a superclass of `Employee` (e.g. `Person`) as the type of the `sendEmail` method. This can sometimes effectively reduce the stamp coupling; but using an interface is usually a more flexible solution.

The second way to reduce stamp coupling is illustrated as follows:

```
public class Emailer
{
  public void sendEmail(String name, String email, String text)
  {...}
 ...
}
```

In this case the stamp coupling has been replaced with data coupling, discussed below.

Data coupling This occurs whenever the types of method arguments are either primitive or else simple classes such as `String`. Methods must obviously have arguments,

therefore some data coupling or stamp coupling is unavoidable. However, you should reduce coupling by not giving methods unnecessary arguments.

The more arguments a method has, the higher the coupling. This is because each caller to the method must have code to prepare the data for each argument; and any changes to how the method declares or interprets each argument may require changes to each caller's code.

There is a trade-off between data coupling and stamp coupling. In the case of a single argument, data coupling is considered looser, and therefore better, than stamp coupling. However, if you replace a single complex argument (stamp coupling) with many simple arguments (data coupling), the total resulting coupling will be higher. In the above code, it was acceptable to eliminate stamp coupling at the expense of adding one extra argument to the `sendEmail` method. It would not have been acceptable to add three or four extra arguments; in such a case, sticking with the stamp coupling (using the `Addressee` interface) would have been better.

Routine call coupling This occurs when one routine (or method in an object-oriented system) calls another. The routines are coupled because they depend on each other's behavior, and the caller depends on the interface of the called routine.

Routine call coupling is always present in any system. However, if you use a sequence of two or more methods to compute something, and this sequence is used in more than one place, then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

For example, imagine that to use a graphics package, you had to write the following sequence of code over and over again:

```
aShape.drawBackground();
aShape.drawForeground();
aShape.drawBorder();
```

You would be better off creating a new method that encapsulated this sequence. Should the arguments of the above three methods ever change, the maintainer would now only have to change your encapsulated method.

Type use coupling This occurs when a module uses a data type defined in another module. Type use coupling naturally occurs in typed languages such as Java. It occurs any time a class declares an instance variable or a local variable as having another class for its type.

Type use coupling is similar to common coupling, but instead of data being shared, only data types are shared. The impact of sharing data types is normally less than the impact of sharing data, hence type use coupling is considered less problematic than common coupling.

The consequence of type use coupling is that if the type definition changes, then the users of the type may well have to change.

Stamp coupling is closely related to type use coupling, therefore the techniques for reducing stamp coupling can also be applied to type use coupling. In particular, you should declare the type of a variable to be the most general

possible class or interface that contains the required operations. For example, when creating a variable that is to contain a collection, you should normally declare its type to be `List`, that is, any class that implements the `java.util.List` interface. The actual instance stored in the variable could be an `ArrayList`, `LinkedList` or `Vector`, or perhaps some other class to be defined later. However, declaring the type to be `List` is sufficient since all the important operations are defined in that interface. The benefit is that your code would be less likely to need to change were you to later decide to use a different type of collection.

Inclusion or import coupling

Import coupling occurs when one component imports a package (as in Java); inclusion coupling occurs when one component includes another (as in C++). Doing this means that the including or importing component is now exposed to everything in the included or imported component – even if it is not actually using the facilities of that component. If the included or imported component changes something on which the includer relies, or adds something that raises a conflict with something in the includer, then the includer must change.

The bigger the imported or included component, the worse the coupling. However, importing a standard package (e.g. one delivered with the programming language) is better than importing a homemade package.

Some inclusion or import coupling is necessary – since it enables you to use the facilities of libraries or other subsystems. However, it is important not to import packages or classes that you do not need: in addition to having to worry about changes to the things you are using, you then also have to worry about changes to things you don't use. For example, your system might suddenly fail if a new item is added to an imported file, and this new item has the same name as something you have already defined in your subsystem.

External coupling

This occurs when a module has a dependency on such things as the operating system, shared libraries or the hardware. It is best to reduce the number of places in the code where such dependencies exist.

The Façade design pattern can reduce external coupling by providing a very small interface to external facilities.

Exercises

E175 Another way to resolve control coupling in `handleMessageFromClient` would be to use Java's reflection mechanism. This permits Java to directly treat a string as a method name, and then to call the method. Investigate reflection, and determine what changes would be required to `handleMessageFromClient`. Then discuss whether the use of reflection would actually be a good design decision, as opposed to keeping the control coupling.

E176 Categorize the following aspects of a design by the types of *coupling* they exhibit.

(a) Class `CourseSection` has public class variables called `minClassSize` and `maxClassSize`. These are changed from time to time by the university administration. Many methods in classes `Student` and `Registration` access these variables.

(b) A user interface class imports a large number of Java classes, including those that draw graphics, those that create UI controls and a number of other utility classes.

(c) A system has a class called `Address`. This class has four public variables constituting different parts of an address. Several different classes, such as `Person` and `Airport` manipulate instances of this class, directly modifying the fields of addresses. Also, many methods declare one of their arguments to be an `Address`.

E177 Describe ways to reduce the cases of coupling described in the last exercise.

E178 What forms of coupling are present in the SimpleChat system? Describe any ways in which coupling can be reduced.

9.5 Software architecture

Architecture plays a central role in building construction. A building's architecture is described using a set of plans that, taken together, represent all aspects of the building. It describes the building from such viewpoints as electricity, plumbing, structure, etc.

The architect is the person in charge of the whole project. He or she has the responsibility to make sure that the building will be solid, cost-effective and satisfactory to the client.

Software architecture is similar. It plays a central role in software engineering, and involves the development of a variety of high-level views of the system. Furthermore, individuals called software architects often lead a team of other software engineers.

Definition: *software architecture* is the process of designing the global organization of a software system, including dividing software into subsystems, deciding how these will interact, and determining their interfaces.

The term 'software architecture' is also applied to the documentation produced as a result of the process. For clarity, this documentation is often also called the *architectural model*.

The importance of developing an architectural model

Software engineers discuss all aspects of a system's design in terms of the architectural model. Decisions made while this model is being developed therefore have a profound impact on the rest of the design process. The architectural model is the core of the design; therefore all software engineers need to understand it.

The architectural model will often constrain the overall efficiency, reusability and maintainability of the system. Poor decisions made while creating this model will constrain subsequent design.

There are four main reasons why you need to develop an architectural model:

- **To enable everyone to better understand the system**. As a system becomes more and more complex, making it understandable is an increasing challenge. This is especially true for large, distributed systems that use sophisticated technology. A good architectural model allows people to understand how the system as a whole works; it also defines the terms that people use when they communicate with each other about lower-level details.
- **To allow people to work on individual pieces of the system in isolation**. The work of developing a complex software system must be distributed among a large number of people. The architecture allows the planning and co-ordination of this distributed work. The architecture should provide sufficient information so that the work of the individual people or teams can later on be integrated to form the final system. It is for that reason that the interfaces and dynamic interactions among the subsystems are an important part of the architecture.
- **To prepare for extension of the system**. With a complete architectural model, it becomes easier to plan the evolution of the system. Subsystems that are envisioned to be part of a future release can be included in the architecture, even though they are not to be developed immediately. It is then possible to see how the new elements will be integrated, and where they will be connected to the system. Architects designing buildings often use this technique – their drawings show not only the proposed building but also its future extensions (Phase I, Phase II, etc.). Specialists like electrical engineers can then plan the cabling to take into account the future needs of the foreseen extension.
- **To facilitate reuse and reusability**. The architectural model makes each system component visible. This is an important benefit since it encourages reuse. By analyzing the architecture, you can discover those components that can be obtained from past projects or from third parties. You can also identify components that have high potential reusability. Making the architecture as generic as possible is a key to ensuring reusability.

Contents of a good architectural model

A system's architecture will often be expressed in terms of several different *views*. These can include:

- The logical breakdown into subsystems. This is often shown using package diagrams, which we will describe later. The interfaces among the subsystems must also be carefully described.
- The dynamics of the interaction among components at run time, perhaps expressed using interaction or activity diagrams.
- The data that will be shared among the subsystems, typically expressed using class diagrams.

- The components that will exist at run time, and the machines or devices on which they will be located. This information can be expressed using component and deployment diagrams, which are discussed later.

An important challenge in architectural modeling is to produce a relevant and synthetic picture of a large and complex system. In other words, the reader should be able to understand the system very quickly by looking at the different views. To enable this, it should be clear how the views relate to each other.

To ensure the maintainability and reliability of a system, an architectural model must be designed to be *stable*. Being stable means that the new features can be easily added with only small changes to the architecture.

When developing custom software, the architecture should be expressed clearly enough that it can be used to communicate effectively with clients. The clients may not need to know other details of the design. However, they often want to understand the architecture so that they can be confident the software is being designed well, and can monitor development progress. The architectural diagrams used for the construction of buildings are also used to communicate with clients.

How to develop an architectural model

The system's architecture must take its overall shape very early in the design process, although it will continue to mature as iterative development proceeds.

The basis for the architectural model will be the system domain model and the use cases. The first draft of the architectural model should be created at the same time as these. These give the architect an idea about which components will be needed and how they will interact. At the same time, the early architecture will give use case modelers guidance about the steps the user will need to perform. For example, if the use cases describe a process of accessing information that is stored centrally in some repository, this guides the architect to think in terms of a client–server architecture. Similarly, the fact that there is a client–server architecture guides the use case modeler to add use cases for logging in, account creation, etc.

The following are some steps that you can use iteratively as you refine the architecture.

1. Start by sketching an outline of the architecture, based on the principal requirements, including the domain model and use cases. At this stage, you can determine the main components that will be needed, such as databases, particular hardware devices and the main software subsystems. You can also choose among the various architectural patterns we will discuss later, such as using a client–server architecture or a pipe-and-filter architecture. It can be worthwhile having several different teams independently develop a first draft of the architecture; the teams can then meet to pick the best architecture, or to merge together the best ideas.

2. Refine the architecture by identifying the main ways in which the components will interact, and by identifying the interfaces among them. Also, decide how each piece of data and functionality will be distributed among the various components. Now is the time to determine if you can reuse an existing framework. If possible, you might decide to transform your architecture into a generic framework that can be reused by others.
3. Consider each use case, adjusting the architecture to make it realizable. At this stage you try to finalize the interface of each component.
4. Mature the architecture as you define the final class diagrams and interaction diagrams.

Describing an architecture using UML

All UML diagrams can be useful to describe aspects of the architectural model. Remember that the goal of architecture is to describe the system at a very high level, with emphasis on software components and their interfaces. Use case diagrams can provide a good summary of the system from the user's perspective. Class diagrams can be used to indicate the services offered by components and the main data to be stored. Interaction diagrams can be used to define the protocol used when two components communicate with each other.

In addition to the UML diagrams we have already studied in this book, three other types of UML diagram are particularly important for architecture modeling: package diagrams, component diagrams and deployment diagrams. These are used to describe different aspects of the organization of the system. In the next three subsections we will survey the essentials of these types of diagram.

Packages

Breaking a large system into subsystems is a fundamental principle of software development. A good decomposition helps make the system more understandable and therefore facilitates its maintainability.

In UML, a *package* is a collection of modeling elements that are grouped together because they are logically related. Note that a UML package is not quite the same thing as a Java package, which is a collection containing only classes. However, a very common use of UML packages is to represent Java packages.

A package in UML is shown as a box, with a smaller box attached above its top left corner. The packages of the SimpleChat system are illustrated in Figure 9.6. Inside the box you can put practically anything, including classes, instances, text or other packages.

When you define a package, you should apply the principles of cohesion and coupling discussed earlier. Increasing cohesion means ensuring that a package only has related classes; decreasing coupling means decreasing the number of dependencies as much as possible.

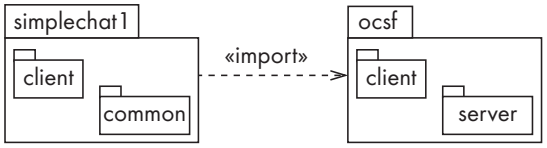

Figure 9.6 An example package diagram

You show dependencies between packages using a dashed arrow. A dependency exists if there is a dependency between an *element* in one of the packages and an *element* in another. To use a package, it is required to have access to packages that it depends on. Also, changes made to the interface of a package will require modification to packages that depend on it.

A package that depends on many others will be difficult to reuse, since using it will also necessitate importing its dependent packages. Circular dependencies among packages are particularly important to avoid. Finally, making the interface of a package as simple as possible greatly simplifies its use and testing. The Façade pattern can help to simplify a package interface.

Component diagrams

A component diagram shows how a system's components – that is, the physical elements such as files, executables, etc. – relate to each other. The UML symbol for a component is a box with a little 'plug' symbol in the top-right corner. An example is shown in Figure 9.7; many more examples can be found in the next section.

Figure 9.7 An example component diagram

A component provides one or more interfaces for other components to use. The same 'lollipop' symbol is used for an interface as was introduced in Chapter 5. To show a component using an interface provided by another, you use a semi-circle at the end of a line. Figure 9.7 show how these two symbols plug together.

Various relationships can exist among components, for example:

- A component may *execute* another component, or a method in the other component.
- A component may *generate* another component.
- Two components may *communicate* with each other using a network.

It is easy initially to confuse component diagrams with package diagrams. The difference is that package diagrams show *logical groupings* of design elements, whereas component diagrams show relationships among types of *physical* components.

Deployment diagrams

A deployment diagram describes the hardware where various instances of components reside at run time. An example is shown in Figure 9.8. A node in a deployment diagram represents a computational unit such as a computer, a processing card, a sensor or a device. It appears as a three-dimensional box.

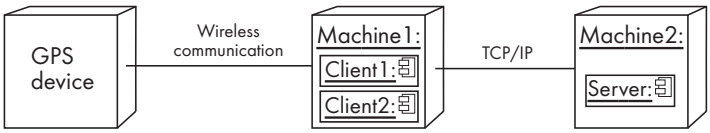

Figure 9.8 **An example deployment diagram**

The links between nodes show how communication takes place. Each node of a deployment diagram can include one or several run-time software components. Various artifacts such as files can also be shown inside nodes.

9.6 Architectural patterns

The notion of patterns, introduced in Chapter 5, can be applied to software architecture. In this chapter we present several of the most important *architectural patterns*, which are also often called *architectural styles*. Each allows you to design flexible systems using components that are as independent of each other as possible.

The Multi-Layer architectural pattern

Building software in layers is a classical architectural pattern that is used in many systems. It is so important that layer cohesion was one of the types of cohesion we presented earlier when discussing Design Principle 2.

As we discussed, in layered systems each layer communicates only with the layers below it – in many cases, only the layer immediately below it. Each layer has a well-defined API, defining the services it provides.

A complex system can be built by superimposing layers at increasing levels of abstraction. The Multi-Layer architectural pattern makes it possible to replace a layer by an improved version, or one with a different set of capabilities.

It is particularly important to have a separate layer at the very top to handle the user interface. Independence of the UI layer allows the application to have several different UIs. These could be UIs running on different platforms, UIs for 'professional' versus 'standard' versions of the application, or UIs designed for different locales (discussed in Chapter 7). We will look at the UI layer in more detail below when we discuss the Model–View–Controller architectural pattern.

Layers immediately below the UI layer provide the application functions determined by the use cases. Layers at the bottom provide services such as data storage and transmission. This is illustrated in Figure 9.3(a).

Patterns at different levels of abstraction

Patterns can be created for any activity involving human expertise. In software engineering, they occur at many levels of abstraction.

At the lowest level are programming *idioms*; these describe preferred ways to solve detailed programming problems, and are out of the scope of this book. Moving up the abstraction scale are the *design patterns* such as Delegation and Observer, and the *modeling patterns* such as Abstraction–Occurrence presented in Chapter 6. At the top of the abstraction scale are the *architectural patterns* discussed here.

Most operating systems are built according to the Multi-Layer architectural pattern, as shown in Figure 9.3(b). A low-level *kernel* layer deals with such functions as process creation, swapping, and scheduling. Higher-level layers deal with such functions as user account management, screen display, etc. The layers are often further subdivided into smaller subsystems.

Most communications systems exhibit a layered architecture. A simplified illustration of this is shown in Figure 9.3(c). At the bottom level, there are facilities for transmitting and receiving signals. Above this is a layer that deals with splitting messages into packets and reconstructing messages that are received. Still higher is a layer that deals with handling ongoing connections with a remote host (e.g. using sockets). At the top is a layer that handles various protocols, such as http, used by application programs.

The SimpleChat system uses a layered architecture to separate the user interface from the core of the system. For example, the class `ChatClient` is separated from the class `ClientConsole`. The Observable layer of the OCSF allows the core of a client or server to be further separated into a layer that has the application logic, and one that deals with client–server communication.

Although the normal assumption is that the communication between layers will be by procedure calls, it can also be performed in some systems using inter-process communication. In other words, the lower layers can become servers and the higher layers can become clients. This illustrates how the Multi-Layer and Client–Server architectural patterns (discussed next) can be used together. If we did use inter-process communication to implement a layered architecture, we would typically redraw Figure 9.3 using a component diagram.

The Multi-Layer architectural pattern helps you adhere to many of the design principles discussed earlier, in particular:

1. *Divide and conquer*. The separate layers can be independently designed.
2. *Increase cohesion*. Well-designed layers have layer cohesion; in other words, they contain all the facilities to provide a set of related services, and nothing else.
3. *Reduce coupling*. Well-designed lower layers do not know about the higher layers. The higher layers can therefore be replaced without impacting the lower layers. Also, the only connection between layers is through the API.
4. *Increase abstraction*. When you design the higher layers, you do not need to know the details of how the lower layers are implemented. This makes designing high-level facilities much easier.

5. *Increase reusability.* The lower layers can often be designed generically so that they can be used to provide the same services for different systems.
6. *Increase reuse*. You can often reuse layers built by others that provide the services you need – a layer for loading and storing from a database, for example.
7. *Design for flexibility.* Designing in layers gives you the flexibility to add new facilities that build on lower-level services, or to replace higher-level layers.
8. *Anticipate obsolescence*. Databases and UI systems tend to change; by isolating these in separate layers, the system becomes more resistant to obsolescence.
9. *Design for portability.* All the facilities that are dependent on a particular platform can be isolated in one of the lower layers.
10. *Design for testability.* Individual layers, particularly the UI layer, database layer and communications layer, can be tested independently.
11. *Design defensively.* The APIs of layers are natural places to build in rigorous checks of the validity of inputs. You can design a system so that if a higher layer fails, the lower layers continue to run. The opposite can also be made true; for example, if a database layer crashes, it could be made to restart automatically.

The Model–View–Controller (MVC) architectural pattern

Model–View–Controller, or MVC, is an architectural pattern used to help separate the user interface layer from other parts of the system. Not only does MVC help enforce layer cohesion of the user interface layer, but it also helps reduce the coupling between that layer and the rest of the system, as well as between different aspects of the UI itself.

The MVC pattern separates the functional layer of the system (the *model*) from two aspects of the user interface, the *view* and the *controller*. This is illustrated in Figure 9.13. Although the three components are normally

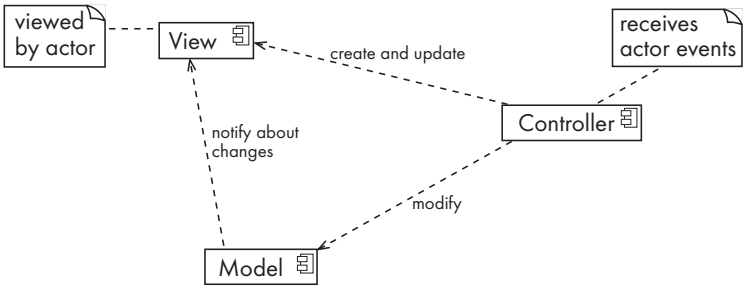

Figure 9.13 The Model–View–Controller (MVC) architectural pattern for user interfaces

instances of classes, we use a component diagram to emphasize the fact that the components could also be separate threads or processes.

The *model* contains the underlying classes whose instances are to be viewed and manipulated.

The *view* contains objects used to render the appearance of the data from the model in the user interface. The view also displays the various controls with which the user can interact.

The *controller* contains the objects that control and handle the user's interaction with the view and the model. It has the logic that responds when the user types into a field or clicks the mouse on a control.

The model does not know what views and controllers are attached to it. In particular, the Observer design pattern is normally used to separate the model from the view. The MVC architectural pattern therefore exhibits layer cohesion and is a special case of the Multi-Layer architectural pattern.

MVC in web architectures

Web architectures generally use MVC as follows:

1. The View component generates html for display by the browser.
2. There is a component that interprets http 'post' transmissions coming back from the browser – this is the Controller.
3. There is an underlying system for managing the information – this is the Model.

Examples of View generation technology are JSP and ASP – you will often see web pages that have these suffixes. These technologies start with an outline of an html document and use programs to fill in missing pieces. Each time the page is displayed, it can therefore have different content (at the very least, advertisements are often inserted). The original Controller technology was called 'CGI' (Common Gateway Interface); there are now, however, many others available.

The View and Controller technologies can be combined: a program that interprets a 'post' transmission will often then generate a new page. The model, however, should always be kept separate.

Sometimes no specific controller component is created – the most important aspect of the MVC pattern is separation of the model and the view. The term model-view separation is used to describe this situation.

The MVC architectural pattern allows us to adhere to the following design principles:

1. *Divide and conquer*. The three components can be somewhat independently designed.

2. *Increase cohesion*. The components have stronger layer cohesion than if the view and controller were together in a single UI layer.

3. *Reduce coupling*. The communication channels between the three components are minimal and easy to find.

6. *Increase reuse*. The view and controller normally make extensive use of reusable components for various kinds of UI controls. The UI, however will become application specific, therefore it will not be easily reusable.

7. *Design for flexibility*. It is usually quite easy to change the UI by changing the view, the controller, or both.

10. *Design for testability*. You can test the application separately from the UI.

MVC in Java

Several of the Swing GUI components of the Java API are based on the MVC architectural pattern.

For example, `JList` (the view) is a graphical component used to visualize a list of items. This list is the visual representation of data handled by an `AbstractListModel` (the model). It contains the data and notifies the `JList` whenever a change is made to the data by the application. These two classes follow the Observer pattern (see Chapter 6); the JList registers itself as an observer of the `AbstractListModel` instance as follows (note that in the Java API, observers are usually called *listeners*):

```
listModel.addListDataListener(jList)
```

Finally, each time a modification is made to the list, the method `fireIntervalAdded` or `fireIntervalRemoved` is called by the controller.

9.7 Writing a good design document

Design documents serve two main purposes. Firstly, they help you, as a designer or a design team, to *make good design decisions*. The process of writing down

your design helps you to think more clearly about it and to find flaws in it. Secondly, they help you *communicate the design* to others.

We will now examine both of these purposes.

Design documents as an aid to making better designs

Design documents help you, as a designer, because they force you to be explicit and to consider the important issues before starting implementation. They also allow a group of people to review the design and therefore to improve it.

There has been a tendency among software developers to omit design documentation or to document the design only *after* it is complete. In other branches of engineering, doing this has always been considered completely unacceptable: engineers know that it would lead to serious mistakes and in most cases would make planning for construction impossible. For example, without creating plans in advance of building construction it would be impossible to know what building supplies to order, and nobody would be able to review the design to ensure it adheres to standards, such as having adequate fire exits.

Unfortunately, because software is intangible, it is sometimes *possible* to jump directly into programming without any design documents. This does not, however, make doing so a good idea. Plunging directly into programming without writing a design document tends to result in an inflexible and overly complex system.

Design documents as a means of communication

When writing anything, it is important to know the *audience* for your work. Design documents are used to communicate with three groups of individuals. In general, you can expect most documents to be read by all three groups:

- Those who will be *implementing* the design, that is, the programmers.
- Those who will need, in the future, to *modify* the design.
- Those who need to create systems or subsystems that *interface* with the system being designed.

Knowing that these are the audiences for the design document can help the designer decide what information to include. For example, to communicate with designers of other systems, you can make explicit the services that your system provides and how to use them. To communicate with future maintainers, you can give a high-level overview of your design to help them understand it, and explain areas where your design was made flexible to allow for enhancement.

It is crucial to not only include the design as it exists, but also to include the *rationale* for the design: that is, the reasoning you used when making your design decisions. Providing the rationale allows the reader to better understand the design. It also allows reviewers to determine whether good decisions were

made, and helps the maintainers determine how to change the design. A maintainer may be tempted to change the design so that it reflects one of the alternatives you rejected. Knowing your reasoning means that the maintainer will not do this capriciously, but with the benefit of your prior insight.

Contents of a design document

We suggest that a design document should contain the following information. As with all the documentation types described in this book, you should use this as a general guide only. Each company should have specific formats you need to follow so as to be consistent with other design documents. You will also need to vary the kinds of information in each section depending on the kind of design (e.g. architectural or detailed) that you are producing.

A. **Purpose**. Specify what system or part of the system this design document describes. Make reference to the requirements that are being implemented by this design – doing so ensures that there is *traceability* from the requirements to the design.

B. **General priorities**. Describe the priorities used to guide the design process. For example, how important was maintainability or efficiency as the design was being prepared?

C. **Outline of the design**. Give a high-level description of the design that allows the reader to get a general feeling for it quickly. A diagram can often serve this purpose.

D. **Major design issues**. Discuss the important issues that had to be resolved. Give the possible alternatives that were considered, the final decision and the rationale for the decision.

E. **Details of the design**. Give any other details the reader will need to know that have not yet been mentioned. These might include detailed descriptions of the protocol used to communicate between client and server, the overviews of data structures and algorithms, as well as how to use various APIs.

In general, when writing a design document, ensure that it is neither too short nor too long. In keeping the document at the right length, be guided by the intended audience; ensure that the information the readers will need to learn is readily available. At the same time, remember that there is no point writing information that would never be read because the reader already knows it or can easily find it from some other source. In particular:

- Avoid documenting information that would be readily obvious to a skilled programmer or designer.
- Avoid writing details in a design document that would be better placed as comments in the code.

- Avoid writing details that can be extracted automatically from the code, such as the list of public methods.

The latter two points deserve some elaboration. When you are doing design, you can actually create skeletons for the files that will contain the code. You write in these skeletons some of the high-level code comments as well as the templates for public methods. If instead you were to write this material in design documents, then when you transfer it to the code, you would have to maintain the information in both places, since it will inevitably change.

9.8 Design of a feature for the SimpleChat instant messaging application

This section presents a short design document that extends the detailed requirements example that was presented in Section 4.12.

A. **Purpose.** This document describes important aspects of the implementation of the `#block`, `#unblock`, `#whoiblock` and `#whoblocksme` commands of the SimpleChat system.

For the requirements, see Section 4.12.

B. **General priorities.** Decisions in this document are made based on the following priorities (most important first): Maintainability, Usability, Portability, Efficiency.

C. **Outline of the design.** Blocking information will be maintained in the `ConnectionToClient` objects. The various commands will update and query the data using `setValue` and `getValue`.

D. **Major design issues.**

Issue 1: Where should we store information regarding the establishment of blocking?

Option 1.1: Store the information in the `ConnectionToClient` object associated with the client requesting the block.

Option 1.2: Store the information in the `ConnectionToClient` object associated with the client that is being blocked.

Decision: *Point 2.2 of the specification requires that we be able to block a client even if that client is not logged on. This means that we must choose option 1.1 since no* `ConnectionToClient` *will exist for clients that are logged off.*

E. **Details of the design:**

Client side:

- The four new commands will be accepted by `handleMessageFromClientUI` and passed unchanged to the server.

- Responses from the server will be displayed on the UI. There will be no need for `handleMessageFromServer` to understand that the responses are replies to the commands.

Server side:

- Method `handleMessageFromClient` will interpret `#block` commands by adding a record of the block in the data associated with the originating client. This method will modify the data in response to `#unblock`.
- The information will be stored by calling `setValue("blockedUsers", arg)` where `arg` is a `Vector` containing the names of the blocked users.
- Method `handleMessageFromServerUI` will also have to have an implementation of `#block` and `#unblock`. These will have to save the blocked users as elements of a new instance variable declared thus: `Vector blockedUsers;`
- The implementations of `#whoiblock` in `handleMessageFromClient` and `handleMessageFromServerUI` will straightforwardly process the contents of the vectors.
- For `#whoblocksme`, a new method will be created in the server class that will be called by both `handleMessageFromClient` and `handleMessageFromServerUI`. This will take a single argument (the name of the initiating client, or else '`server`'). It will check all the `blockedUsers` vectors of the connected clients and also the `blockedUsers` instance variable for matching clients.
- The `#forward`, `#private` and simple message commands will be modified as needed to reflect the specifications. Each of these will each examine the relevant `blockedUsers` vectors and take appropriate action.

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.

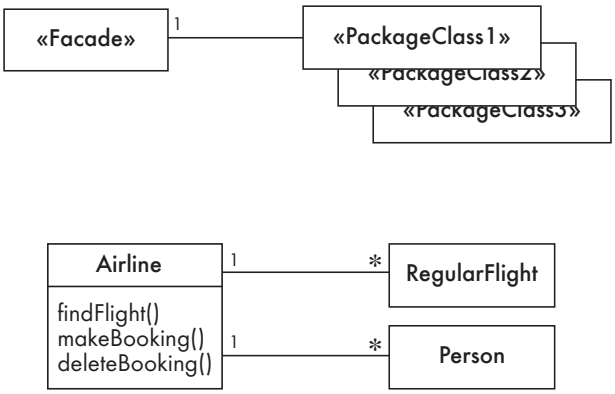

Figure 6.11 Template and example of the Façade design pattern

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.

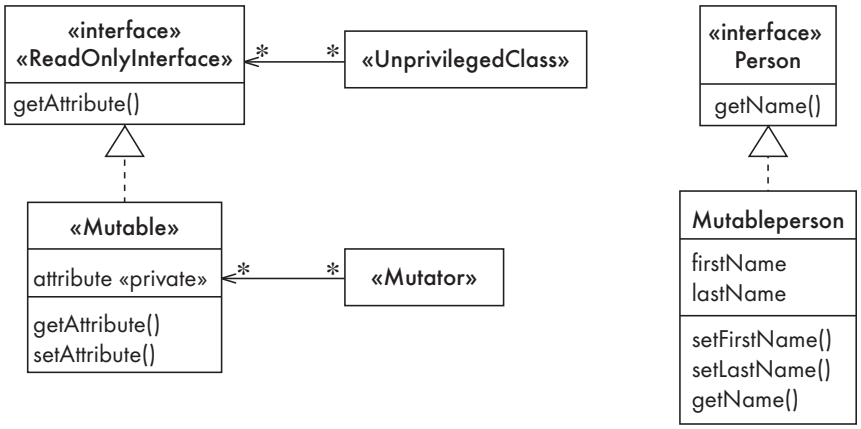

Figure 6.12 Template and example of the Read-Only Interface design pattern

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).