9 Architecting and designing software

In earlier chapters you have seen how to gather requirements and build models of software systems. We have also looked at how to design the external functionality of the system – the user interface. In this chapter, we will look at general principles of design, as well as various patterns that can be used to construct the high-level architecture of a system. We will place particular emphasis on communicating design decisions to others.

In this chapter you will learn about the following

- Design as a series of design decisions.
- Various approaches and types of design, including top-down design starting with the architecture, and bottom-up design starting with utilities.
- Design principles that lead to maintainable software, such as 'divide and conquer', striving for high cohesion and low coupling, as well as using good abstractions to hide details, thus simplifying the system.
- How to perform design while keeping portability, reuse, reusability and testability in mind.
- Evaluating trade-offs among alternatives, and applying basic cost–benefit analysis when making design decisions.
- Software architectures for high-level design.
- How to write a good design document.

9.1 The process of design

We start this chapter by taking a very high-level view of design, attempting to answer the following questions. What exactly is design? What general approaches do designers use? What types of design are there?

Design as a series of decisions

Definition: in the context of software, *design* is a problem-solving process whose objective is to find and describe a way to implement the system's functional requirements, while respecting the constraints imposed by the quality, platform and process requirements (including the budget and deadlines), and while adhering to general principles of good quality.

A designer is faced with a series of *design issues*, which are sub-problems of the overall design problem. Each issue normally has several alternative solutions, also known as design *options*. The designer makes a *design decision* to resolve each issue – this process involves choosing what he or she considers to be the best option from among the alternatives. To make each design decision, the software engineer uses all the knowledge at his or her disposal, including:

- knowledge of the requirements;
- knowledge of the design as created so far;
- knowledge of the technology available;
- knowledge of software design principles and 'best practices'; and
- knowledge about what has worked well in the past.

Once a decision is made, new issues are raised.

Sometimes there is no suitable choice to resolve a given issue. In that case, the designer may have to go back and revise previous decisions, or else propose that the requirements be changed.

There may not be a single best alternative when dealing with a particular design issue. Several different alternatives may have opposite advantages and disadvantages, with no clear 'winner'. For example, a thin-client system might result in software that is simpler, whereas a fat-client system might result in software that makes more efficient use of CPU and network resources. Both might seem equally good.

Also, when evaluating alternatives, different designers have different knowledge and ideas, therefore they will tend to reach different conclusions. As a result, two designers will rarely come up with exactly the same solution.

The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*. Figure 9.1 illustrates this idea. The

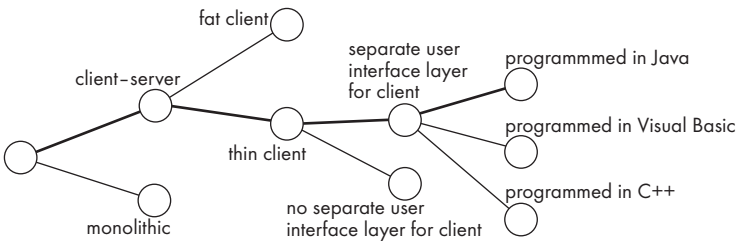

Figure 9.1 **Part of a design space, showing alternative designs resulting from different choices when making design decisions. The lines represent options; the bold path is the set of decisions made**

final software product is the result of all the design decisions made; different sets of decisions lead to a different product.

You should try to explore different paths through design space – investigating the consequences of choosing different alternatives when major issues arise. In Chapter 7, we recommended employing parallel design to do this. In parallel design, several different designers create their own designs. Following parallel design, you analyze the resulting designs to determine which combination of choices seems best for the final product.

We also showed you the process of selecting among alternative designs in Chapter 2, where several exercises asked you to choose among alternative designs of the `PointCP` class. In Chapter 5, we studied alternative ways of modeling problems using class diagrams.

Although there might be several ways to produce a good quality system, some design issues are critical. For these critical decisions, bad choices will lead to a poor system. For example, imagine that very early on in your design you chose not to separate the user interface from the rest of the system. Such a decision would greatly constrain your ability to achieve a flexible, maintainable design. For example, it would become much harder to provide a web-enabled version of your system or to internationalize it.

Each design decision should be recorded, along with the reasoning that went into making the decision (known as the *design rationale*). The entire record of the series of decisions becomes a design document. We will discuss the structure of design documents later in this chapter.

Parts of a system: subsystems, components and modules

It is now time to define some important terms that will help in further discussion. These terms are often used interchangeably, but in this book we will give them the meanings below. The first two terms, *component* and *module*, describe concrete, implemented parts of a system.

Component any piece of software or hardware that has a clear role and can be isolated, allowing you to replace it with a different component with equivalent functionality. Many components are designed to be reusable, but in other cases

they will perform special-purpose functions (e.g. providing the user interface for a particular system). A framework, discussed in Chapter 3, is a kind of component. Other components include source code files, executable files, dynamic link libraries (DLLs) and databases. Some components, such as source files, only exist at compile time; other components, such as certain data files, may only exist at run time.

Module a component that is defined at the programming language level. For example, methods, classes and packages are modules in Java. The modules in the C programming language are files and functions.

The next two terms describe entities that may be implemented in different ways and are therefore more abstract than components and modules.

System a logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both. A system can have a specification that is then implemented by a collection of components. The notion of system can be extended beyond hardware and software to include also people, business processes, organizations, or natural phenomena that work together to achieve something. We will, however, use the more restricted meaning unless otherwise stated. A system continues to exist, even if its components change over the course of time, or are replaced by equivalent components.

Subsystem a system that is part of a larger system, and which has a definite interface. Java uses packages to implement subsystems; individual classes may also implement particular low-level subsystems.

In software engineering, the requirements analysis process determines the responsibilities of a system. The specification process determines the interface of any component built to implement a system. The design process determines how components will be implemented.

Figure 9.2 is a domain model that shows how the above terms are related. Note that you may find these terms used in slightly different ways in other books. Later on in this chapter we will discuss UML's notation for components.

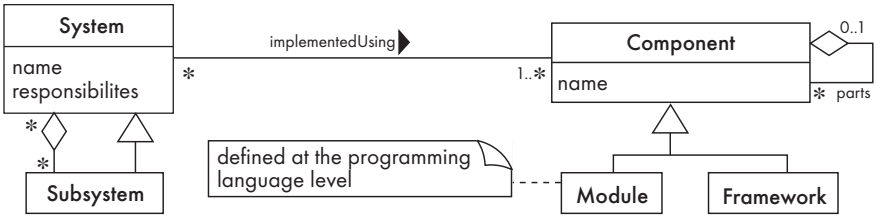

Figure 9.2 **Domain model explaining the concepts of system, subsystem, component and module as used in this book**

Note that for much of the discussion that follows, we talk about designing modules. However, the points we make would be the same if we talked in the context of larger-scale components, or more abstract subsystems.

Top-down versus bottom-up design

There are several fundamentally different sequences with which you can make design decisions. In *top-down design*, you start with the very high-level structure of the system. You then gradually work down towards detailed decisions about low-level constructs. Examples of high-level issues that are approached first in top-down design include the software architecture and the kind of database that will be used. After many higher-level decisions are made, you finally arrive at detailed decisions such as the format of particular data items, and the individual algorithms that will be used.

The inverse approach, *bottom-up design*, involves first making decisions about reusable low-level utilities and then deciding how these will be put together to create high-level constructs.

A mix of top-down and bottom-up design is normally used. Top-down design is almost always needed to give the system a good structure. On the other hand, some bottom-up design helps ensure that you create reusable components that can be used in several places in the overall system.

We will revisit the distinction between top-down versus bottom-up in the next chapter, in the context of testing.

Special types of design

There are many different aspects of software design, including:

- *Architecture design*: the division of software into subsystems and components, as well as the process of deciding how these will be connected and how they will interact, including determining their interfaces. We will talk more about this later in the chapter. Architecture design is commonly referred to as 'software architecture', although the latter term can also refer to the documentation produced, and the whole field of study.
- *Class design*: the design of the various features of classes such as associations, attributes, interactions and states. We discussed essential aspects of this in Chapters 5 and 8.
- *User interface design*, discussed in Chapter 7.
- *Database design*: the design of how data is persistently stored so that it may be accessed by many programs and users, over an indefinite period of time.
- *Algorithm design*: the design of computational mechanisms.
- *Protocol design*: the design of communications protocols – the languages with which processes communicate with each other over a network.

Each of these types of design is the subject of specialized books.

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.

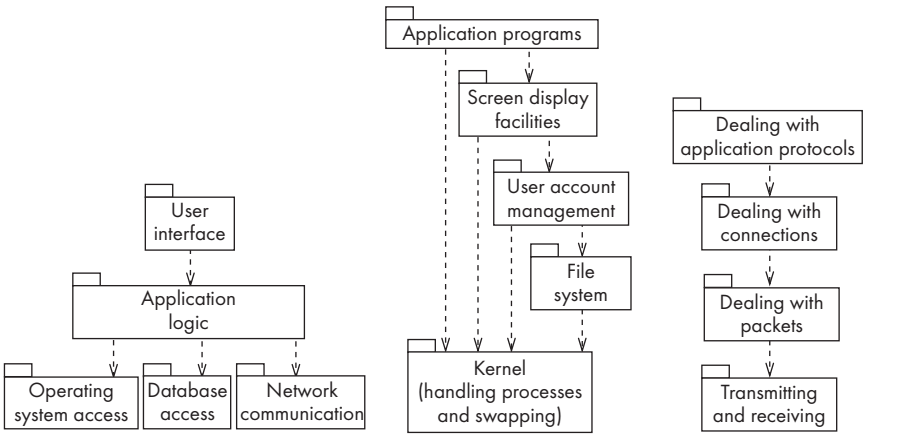

Figure 9.3 **Examples of the use of layers. Higher layers can call on the services of lower layers, but not the other way around**

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.

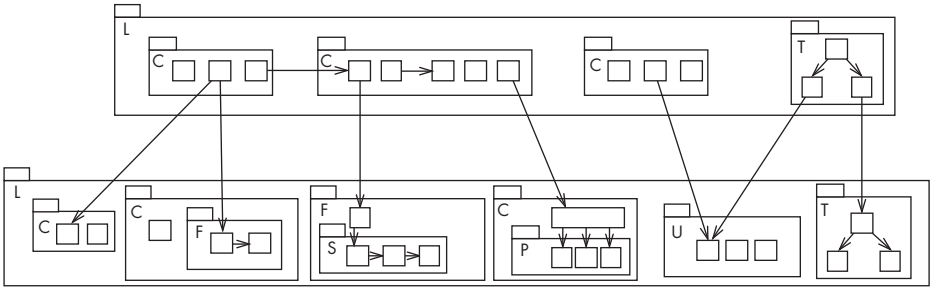

Figure 9.4 **Cohesive modules, nested inside each other, using the services of other modules. The modules are labeled using the first letter of the type of cohesion they represent**

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.

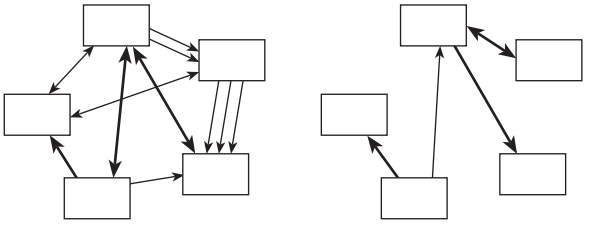

Figure 9.5 **Abstract examples of a tightly coupled system (left) and a loosely coupled system (right). The boldness of the arrows indicates the strength of the coupling**

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.

Design Principle 4: Keep the level of abstraction as high as possible

You should ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity. The general term given to this property of designs is *abstraction*. Abstractions are needed because the human brain can process only a limited amount of information at any one time.

We have discussed many types of abstractions in earlier chapters. In Chapter 2 we introduced procedural abstraction and data abstraction – hiding the details of procedures and data, respectively. In Section 2.7 we discussed several types of abstraction present in object-oriented programs.

Some abstractions, like classes and methods, are supported directly by the programming language. Others, like associations, are present purely in models used by the designer.

Abstractions work by allowing you to understand the essence of something and make important decisions without knowing unnecessary details. The details can be provided in several ways:

- At a later stage of design. For example, when creating class diagrams, you often initially leave out the data types of attributes, and you do not show the implementation details of associations.
- By the compiler or run-time system. For example, dynamic binding takes care of which methods will run.
- By the use of default values. For example, a draw operation that always makes the background white unless some explicit action is taken to change the default.

Design Principle 5: Increase reusability where possible

There are two complementary principles that relate to reuse; the first is to design *for* reuse, and the second is to design *with* reuse. We introduced both of these principles in Chapter 3.

Designing for reusability means designing various aspects of your system so that they can be used again in other contexts, both in your system and in other systems. As discussed in Chapter 3, you can build reusability into algorithms, classes, procedures, frameworks and complete applications. Mechanisms whereby components can be reused include calling procedures and inheriting a superclass.

Important strategies for increasing reusability are as follows:

- Generalize your design as much as possible. As you design a potentially reusable component, imagine several other systems that could use this component. Then design your component so that it could work with the other systems too. For example, if you are creating a facility to draw a particular kind of diagram, why not design it so that it could be used to draw other kinds of diagrams for other applications? Better yet, forget the specific application and focus on the reusable component alone. For example, if you need to create a method to save instances of `Employee` to a binary file, instead consider the problem of saving instances of *any* class to a binary file.
- Follow the preceding three design principles. Increasing cohesion increases reusability since the component has a well-defined purpose. Reducing coupling increases reusability because the component can stand alone. Increasing abstraction increases reusability since abstractions are naturally more general.
- Design your system to contain hooks. As discussed in Chapter 2, a hook is an aspect of the design deliberately added to allow other designers to add additional functionality. One of the barriers to reuse occurs when a component does most of what someone else needs, but not quite everything. If a component has effective hooks, then other people can easily extend it to do what they want. For example, the OCSF system has hooks such as `connectionClosed` that allow application designers to choose to do something interesting when a connection is closed.
- Simplify your design as much as possible. The more complex the component, the less it is likely to be reusable in novel contexts. The most reusable components are those that do one simple thing but do it very well. Basic Unix commands such as `grep`, `cat`, `head`, `tail`, `sort`, `uniq`, `awk`, and `sed` are considered classic examples of reusable components because they are very powerful yet relatively simple. Their simplicity comes from the fact that they all input and output the same data type: streams of characters. Their power comes from the fact that they can be strung together in a large variety of combinations.

There are a number of barriers that tend to thwart attempts to build reusable software. These are discussed in the 'Difficulties and risks' section at the end of Chapter 3.

Design Principle 6: Reuse existing designs and code where possible

Designing with reuse is complementary to designing for reusability. Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components.

Cloning should normally *not* be seen as an effective form of reuse. Cloning involves copying code from one place to another; it should be avoided since, when there are two or more occurrences of the same or similar code in the system, any changes made (e.g. to fix defects) will have to be made in all clones. Unfortunately, maintainers are often not aware of all the clones that exist, and hence only make the change in one place. The bug thus remains, even though the maintainer thinks it is fixed.

In general, it can be acceptable to clone a single line of code; perhaps a line that contains a complicated call to a method with many arguments. However, any time you are tempted to clone more than a couple of lines of code, it is normally best to encapsulate the code in a separate method and call it from all the places it is needed. See the sidebar 'Tolerating Clones?' for a discussion of exceptions to this rule.

Tolerating clones?

Developers are often advised, as we have done here, not to clone code, but rather to create a new routine or method and call it from several places. This remains good general advice, but there are exceptions:

Imagine you have a software system, we will call it 'A', that is thoroughly tested and has a high reliability requirement. Imagine then that you need to create a new system 'B' that is similar to A, but has important differences. The standard advice is to build a framework out of A's code and then create both a new A and a new B built on this framework. But then A would have to be tested all over again since its code would not be identical to the original. The cost of doing this might be higher, in some cases, than the expected cost of having to maintain clones caused by the alternative approach: duplicating A and modifying it to create B. However, if we repeated this cloning process over and over, the cost of maintaining clones would eventually exceed the cost of developing a framework and re-doing the various systems.

In the above situation, of course, if the original designers of A had had the foresight to create a framework in the first place, then the above issue would not arise.

Removing clones that already exist in a reliable, tested system might also often cost more than leaving them there.

Design Principle 7: Design for flexibility

Designing for *flexibility* (also known as *adaptability*) means actively anticipating changes that a design may have to undergo in the future and preparing for them. Such changes might include changes in implementation (e.g. to improve

efficiency or to handle larger volumes of data) or changes in functional requirements.

Ways to build flexibility into a design include:

- Reducing coupling and increasing cohesion. This allows you to more readily replace part of a system. For example, if your current application saves data to a file, you might anticipate that in future you will want to use a commercial database package. Placing the data-saving parts of your system in a subsystem that has layer cohesion will greatly facilitate such a change.
- Creating abstractions. In particular, try to create interfaces or superclasses with polymorphic operations. Doing this allows new extensions to be easily added.
- Not hard-coding anything. Constants should be banished from code. For example, if you want to limit the number of clients that can connect to a server to 25, do not have a line of code that says: `if(numConnections <= 25)…`. Instead, read the maximum value from a configuration file when the server starts. Better yet, make such values preferences that users can change through a preferences dialog.
- Leaving all options open. For example, when a method encounters an exception, it is best that the method throws the exception rather than taking a definite action to handle it. The caller of the method then has the flexibility to decide what to do with the exception.
- Using reusable code and making code reusable. The techniques discussed in the previous two design principles, such as adding hooks, tend also to make designs more flexible.

Cloning objects versus cloning code

In Java there is an interface **`Cloneable`**: many classes implement this, meaning that you can call the method **`clone`** to duplicate instances – that is, to create new, identical, objects.

This is quite different from the 'copying code' sense of cloning discussed here.

Design Principle 8: Anticipate obsolescence

Anticipation of obsolescence is a special case of design for flexibility. Changes will inevitably occur in the technology a software system uses and in the environment in which it runs. Anticipating obsolescence means planning for evolution of the technology or environment so that the software will continue to run or can be easily changed.

The following are some rules that designers can use to better anticipate obsolescence:

- Avoid using early releases of technology. The immediate problem is that early releases are likely to have more defects than later releases. However, even if it is

possible to work around a defect, a secondary problem may then arise: if the provider of the technology fixes the defect in a subsequent release of the technology, the original work-around may no longer work. Even where no defect exists, improvements to the technology, which are especially likely in the first few releases, can render designs that use the technology in need of change. For example, early adopters of Java, in the 1995–1997 time frame, were later required to make many changes to their code because some of the classes and methods they used became *deprecated*. The Java designers declared components to be deprecated when they developed improved designs – they do not intend to support the older components indefinitely, therefore users are forced to update their software.

- Avoid using software libraries that are specific to particular environments. For example, software that makes use of specific features found only in one operating system, or one particular type of hardware, is less likely to be supported in the distant future.

- Avoid using undocumented features or little-used features of software libraries. The little-used features are not only more likely to have defects but, more importantly, the manufacturers may feel that little harm will be done if they are removed or changed. On the other hand, if the technology provider makes changes to heavily used features there will be loud protests from many users, therefore such changes are less likely to be made.

- Avoid using reusable software or special hardware from smaller companies, or from those that are less likely to provide long-term support. A smaller company is more likely to go out of business or not to have the resources to support older versions. It may seem harsh to suggest that we should only trust larger companies – they can go out of business too. However, the probability is higher that a small company will have to abandon a product.

- Use standard languages and technologies that are supported by multiple vendors. Doing this gives you some confidence that important technology will not be orphaned. Many software systems still in existence today are based on obscure proprietary languages. However, standards are not a panacea: they can change, and there may be subtle differences in implementations of a standard that make it difficult to switch to a competing vendor, even if the new vendor ostensibly supports the same standard.

Design Principle 9: Design for portability

Designing for portability shares many things in common with anticipating obsolescence, although the objective is different. Anticipating obsolescence has, as its primary objective, the survival of the software. Design for portability has, as its prime objective, the ability to have the software run on as many platforms as possible, although sometimes this might also be a necessity for survival.

An important guideline for achieving portability is to avoid the use of facilities that are specific to one particular environment. Some programming languages, such as Java, make this easy because the language itself is designed to allow software to run on different platforms unchanged. Nevertheless, even with Java, there can be subtle differences regarding how some features work on different platforms – knowing about these and avoiding them is important. One such difference is class libraries; some companies have produced special Java libraries that work only with that company's compiler, which in turn runs on only one platform. Attempting to port software that uses that library to another platform can be difficult.

Other languages such as C++ have many features that are very much dependent on the particular hardware architecture. You have to be aware, for example, of the order of characters within a word (so-called big-endian versus little-endian), and the number of bits in an integer.

Another important portability issue has to do with text files: the characters used to terminate lines differ from platform to platform.

Design Principle 10: Design for testability

During design you can take steps to make testing easier. Testing, which is the subject of the next chapter, can be performed both manually and automatically. Automatic testing involves writing a program that will provide various inputs to the system in order to test it thoroughly. Therefore it pays to design a system so that automatic testing is made easy.

The most important way to design for testability is to ensure that all the functionality of the code can be executed without going through the graphical user interface. You can achieve this by carefully separating the UI from the functional layer of the system. A test harness can then be written that calls the API of the functional layer. Another good strategy is to provide a command-line version of your system, such as the command-line version of SimpleChat that we presented at the beginning of this book. This will allow you to write a test program that automatically issues commands to your application.

In order to design a Java class for testability you can create a `main` method in each class. Such `main` methods simply exercise the other methods of a class and report any problems.

Design Principle 11: Design defensively

You should never trust how others will try to use a component you are designing. Just like automobile drivers are taught not to trust other drivers, and therefore to *drive* defensively, a software designer should not trust other designers or programmers, and so should *design* defensively. In other words, in order to increase the reliability of your system, you not only need to make sure you don't add any defects yourself, but you must also properly handle all cases where other code attempts to use your component inappropriately.

The most important way to design defensively is to check that all of the inputs to your component are valid. Or, more accurately, check the *preconditions* of each component.

For example, imagine you have a method that determines whether a certain date is a working day. The first thing this method would do is check that the date is valid.

Unfortunately, over-zealous defensive design can result in unnecessarily performing the same validity checks over and over again. For example, imagine the following method:

```
public boolean isWorkingDay(String aDate)
  throws InvalidDateException
{
  if(!isValidDate(aDate)) throw new InvalidDateException();
  return !(isWeekEnd(aDate) || isHoliday(aDate));
}
```

The first line of the method body validates the date. However, due to defensive design, `isWeekEnd` and `isHoliday` may also validate the date. It is a waste of computing power to check the date up to three times like this.

Design by contract is a technique that allows you to design defensively in an efficient and systematic way. The key idea behind design by contract is that each method has an explicit contract with its callers. The contract has a set of assertions that state:

- What *preconditions* the called method requires to be true when it starts executing. The caller has the responsibility to make these preconditions true before making the call.
- What *postconditions* the called method agrees to ensure are true when it finishes executing. The called method has the responsibility to make these postconditions true, before returning.
- What *invariants* the called method agrees will not change as it executes.

Preconditions, postconditions and invariants are all Boolean expressions. If they ever evaluate to false, this indicates that there is a failure. They are similar to the OCL expressions we discussed in Chapter 5; in fact, OCL can be used to write assertions.

Performing assertion checking inside a program is one of the most effective ways to detect and correct errors. Many languages incorporate different mechanisms to write assertions. The ANSI macro `assert(expression)` can be used to this end in C++. In Java, the keyword `assert` has been introduced in version 1.4. For example, in the code below, an explicit precondition assertion has been added, meaning that the method must always be called with a valid date. Not fulfilling this condition would be an error.

```
public boolean isWeekEnd(String aDate)
{
  assert isValidDate(aDate); // precondition.

  return (dayOfTheWeek(aDate)==SUNDAY ||
          dayOfTheWeek(aDate)==SATURDAY);
}
```

Under normal operation, the assertions should not be explicitly evaluated in each method since they have always to be true. However, in the testing phase it is useful to have the assertions explicitly executed, so that you can quickly identify any methods that do not fulfill their contract. In Java, assertions are enabled at compile time by using the `-ea` switch of `javac`. In this mode, `assert` will throw an `AssertionError` if the expression evaluates to `false`. When assertion checking is disabled (default mode), the `assert` statements are simply ignored. You should therefore always switch on assertion checking during development and testing, and turn it off when the system is finally released.

Note that design by contract implies a level of trust. It is like having a driving instructor with you, clarifying the rules of the road, and stopping you from having an accident while you are learning. However, if you make mistakes after getting your license (the assertion checking is off) you can still have an accident. Therefore at the boundaries between major components, such as layers, you should always rigorously check inputs. At these boundaries, you would therefore not use an assertion mechanism that can be turned off.

9.3 Techniques for making good design decisions

The principles discussed in the previous section suggest qualities you should strive to build into software as you design it. In this section we describe two approaches that will help you to make decisions.

Using priorities and objectives to decide among alternatives

Before you start design, you should have established priorities and objectives for various aspects of quality. An objective is a measurable value you wish to attain. A priority states which qualities override others in those cases where you must make compromises.

The qualities to consider when setting priorities and objectives include memory efficiency, CPU efficiency, maintainability, portability and usability. In general, the priorities and objectives should be obtained from the non-functional requirements.

In order to make a design decision, you can perform the following steps:

Step 1 List and describe the alternatives for the design decision.

Step 2 List the advantages and disadvantages of each alternative with respect to your objectives and priorities.

Step 3 Determine whether any of the alternatives prevents you from meeting one or more of the objectives. If it does, you may have to rule it out. However, if none of the alternatives permit you to meet your objectives, you may have to adjust your objectives or else go back to earlier design decisions.

Step 4 Choose the alternative that helps you to best meet your objectives. If several alternatives seem equally good in terms of the objectives, then use the priorities to decide among them.

Step 5 Adjust your priorities for subsequent decision making. If you know you have already met your objectives for some aspects of quality, then you can increase the priority of the other qualities, for which you have not yet met your objectives.

Example 9.1 *You are asked to choose between five different algorithms that can be used to perform a particular distributed financial analysis operation. You are given the following objectives. When everything is otherwise equal, highest priority should be given to the qualities at the top of the list.*

- **Security**. Encryption must not be breakable within 100 hours of computing time on a 2GHz Intel processor, using known cryptanalysis techniques.
- **Maintainability**. No specific objective.
- **CPU efficiency**. Must respond to the user within one second when running on an 800MHz Intel processor (the slowest machine the users are likely to use).
- **Network bandwidth** efficiency. Must not require transmission of more than 8 KB of data per transaction.
- **Memory efficiency**. Must not consume over 30 MB of RAM.
- **Portability**. Must be able to run on Windows 2000 and later versions of Windows, as well as Mac OS and Linux.

You evaluate the algorithms and determine the information in Table 9.3. 'DNMO' means that the algorithm *Does Not Meet the Objective*; the objective is met in the other cases. A dash means that you did not evaluate the factor since you found out that the algorithm did not meet two separate objectives.

We first look at whether the objectives are met. It turns out that none of the algorithms meet the CPU efficiency objective. We are, however, unable to find any algorithms that are better in this respect, therefore we have to lower our standards and choose the algorithms that come closest (A, B and D). Algorithm D, however, does not meet the bandwidth efficiency objective and we therefore rule it out.

Next, we use the remaining priorities to decide between algorithms A and B. Both algorithms are equal in terms of our top priority, security. However, algorithm B is better in terms of the second priority, maintainability; it therefore becomes our final choice.

Table 9.3 **Algorithm evaluations**

	Security	*Maintainability*	*Memory efficiency*	*CPU efficiency*	*Bandwidth efficiency*	*Portability*
Algorithm A	High	Medium	High	Medium; DNMO	Low	Low
Algorithm B	High	High	Low	Medium; DNMO	Medium	Low
Algorithm C	High	High	High	Low; DNMO	High	Low
Algorithm D	—	—	—	Medium; DNMO	DNMO	—
Algorithm E	DNMO	—	—	Low; DNMO	—	—

Exercise

E179 You are given Table 9.4 showing the quality levels achieved by various software architectures.

Table 9.4 **Quality levels achieved by various software architectures**

Software architecture	*Maintain-ability*	*Memory required*	*CPU speed required*	*Bandwidth required*	*Portable to which platforms?*
A	High	20 MB	1 GHz needed	35 Kbps	Unix, Windows
B	High	14 MB	500 MHz needed	1 Mbps	Windows only
C	High	8 MB	2 GHz needed	2 Kbps	Windows, Macintosh
D	Medium	20 MB	1 GHz needed	30 Kbps	Unix only

Determine which architecture you might choose if you had the following objectives and priorities. Justify your answer.

(a) Objectives: runs on Windows; works on a 30 Kbps connection or faster; works on a 1 GHz machine or faster; requires no more than 25 MB memory. General priorities, starting with first: bandwidth efficiency, CPU efficiency, portability, memory efficiency, maintainability.

(b) Same as (a) except bandwidth drops to fourth priority.

(c) Objectives: runs on Unix; works on 1 Mbps connection or faster; works on 500 MHz machine or faster; requires no more than 40 MB memory. General priorities, starting with first: maintainability, portability, bandwidth, CPU speed, memory.

Using cost–benefit analysis to choose among alternatives

An important consideration when you do design is finding ways to reduce costs and increase benefits. Whenever you make a design decision in which the alternatives have different benefits or costs, you can therefore perform cost–benefit analysis to ensure you are making the best decision. You would certainly do this if you are adding an optional aspect of the design, or if the priorities and objectives do not lead to a clear decision.

Cost–benefit analysis is widely taught in management courses, but it is not just for project managers. Individual software engineers should be able to use this technique, since they must make day-to-day design decisions.

You cannot expect to be completely accurate when performing cost–benefit analysis. However, even 'back-of-the-envelope' computations based on rough estimates can help you to make better decisions. Often, it becomes very clear that a certain option will cost far more than an alternative.

To estimate the costs of a new feature or design alternative, you should add up estimates of the following:

- The incremental cost of doing the *software engineering* work, including ongoing maintenance for the life of the system. This includes the work involved in requirements, design, implementation, quality assurance, etc. By 'incremental', we mean the extra cost that would be required if you chose this alternative. The software engineering cost is proportional to the amount of time spent by software engineers, commonly measured in person-days or person-months. Most organizations convert this into monetary terms by multiplying by a factor that accounts for the average salary plus other costs associated with employing a person, such as their office space.
- The incremental costs of any *development technology* that you will have to buy, such as programming languages, reusable components, databases etc.
- The incremental costs that *end-users and product support personnel* will experience. These costs include the extra installation time, training time, help-desk time, learning time and data entry time, as well as extra licenses for reused components, and any extra hardware needed.

In Chapter 11, we will look at other aspects of cost estimation, such as techniques for doing it as accurately as possible.

To estimate the *benefits* of a new feature or design alternative, you should add up the following:

- The incremental software engineering time saved. For example, an improvement in flexibility might considerably reduce maintenance cost.

- The incremental benefits measured in terms of either increased sales or else financial benefit to users. You can base your estimate on increased sales if your product is generic. On the other hand, if the product is custom in nature, then you can estimate how much money users could make or save if you implemented the alternative under consideration. Both figures will depend heavily on making an accurate estimate of the number of users who will eventually use your system.

Example 9.2 *You are the software architect for the GANA system and you are trying to decide whether it would be cost-effective to develop a generic framework for navigation systems. Outline the costs and benefits that should be considered.*

Costs:

- Extra software engineering work: you estimate this will take 3 person-months of extra time to work out the generic requirements, 2 person-months of extra time to do the design, 1 extra person-month for the implementation, and 2 extra person-months for the testing.
- There will be no costs for extra development technology.
- There will be no extra end-user costs.

Benefits:

- Software engineering work saved: you estimate that over the next three years, your company will be developing more advanced navigation systems for the trucking industry and the search and rescue industry. These will, together, save 6 person-months of labor if they can build on your framework. You also estimate that the framework will save you 4 person-months of maintenance time for the GANA system itself.
- You also estimate that the framework will allow you to get new products to market faster. The potential benefit of this is hard to quantify, but it could increase sales by over £200,000.

Even though the above estimates are very approximate, they support your decision to develop a framework. This is true even if you do not consider the uncertain benefits of getting products to market faster.

9.4 Model Driven Development

In Chapters 5, 6 and 8 you learned how to develop models of software using UML. As we mentioned, a model is an abstraction or view of a system that helps you to design it and analyze it. When you study a diagram in a model, what you see should faithfully represent the final system, but only a *simplified* view of an aspect of the system.

Since the earliest days of software engineering, people have used models; and since the mid 1990s these have been typically developed in UML. However, often in the past the models were little more than diagrams – very useful diagrams, but just diagrams. Developers used the diagrams as a guide to their programming. Gradually, various modeling tools were given the capability to *generate* some of the code; programmers could then fill in the missing details.

Now, tools are available which can take models and generate *all* of the code for certain types of applications. The model effectively becomes a form of high-level program and all development work can take place exclusively in the model. This process, known as *model-driven development*, is expected to become more and more widespread over the next 10–20 years as tools become more sophisticated.

Increasing abstraction in programming

In the beginning (the early 1950s) people programmed in *machine language*: directly manipulating the binary op-codes that the CPU uses as its instructions. Then computer scientists had the bright idea of creating *assembly language*: abstracting some of the detail away, making the machine code more human readable using mnemonics for each instruction, and adding higher-level macro capabilities to deal with repetitive tasks. Assemblers translated this into machine language.

The next stage in the evolution of programming was *high-level languages*: this started with Fortran and Cobol, and eventually led to modern languages such as Java, C#, etc. Compilers are programs that initially generated assembler code, and now directly generate machine code or else bytecode for virtual machines. High-level languages added numerous abstractions to make programming simpler: types systems, higher level statements, classes, etc.

Model-driven development simply takes this process further: by modeling in UML the developer can take advantage of abstractions such as states and associations without having to worry about the details of coding them. Tools can generate high-level language code, machine code or bytecode.

Model-driven development does not, however, eliminate the need to code certain types of detailed algorithms. That is why tools that generate entire systems from UML models must also provide a language for implementing the actions and activities that we discussed in Chapter 8. UML provides a lot of detail about the semantics of the actions and activities that must be supported by such a tool; however, it leaves the syntax of an appropriate *action language* to the tool. We will not discuss the details of UML actions any further, except to say that tools can allow the developer to code the actions in a language such as Java.

There are various different approaches to model-driven development. An approach developed by the OMG (who also develop UML) is called *Model Driven Architecture* (MDA).

One of the important features of model-driven development is that the models themselves can be created at several different levels of abstraction. In MDA, one first develops a Platform Independent Model (PIM). This describes the system's data and activities in a very general way. One then adds detail to

create one or more Platform Specific Models (PSMs). For example, you could develop a PIM for a Police Information System; this would contain class diagrams representing all the data to be manipulated, various state diagrams, various activity diagrams, as well as other types of diagrams we will discuss in the next section. This model could, however, be implemented as a web-based system, a client–server system using a technology such as OCSF, something else entirely, or even several of these.

To develop the details for each type of implementation of the Police Information System, you would need a PSM. The PSM for a web-based system would describe the various html pages that would be needed, and the various programs that would interpret http form data generated when the user selects 'submit' on a web page.

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.

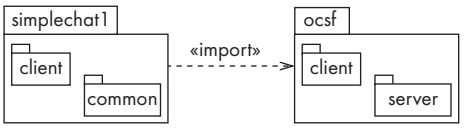

Figure 9.6 **An example package diagram**

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.

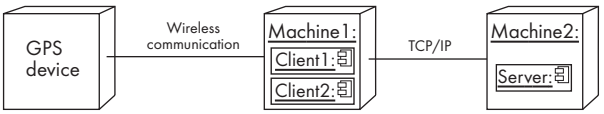

Figure 9.8 **An example deployment diagram**

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.

Exercise

E180 Describe how you might divide a video-game system into layers. Consider that the system has components which do some of the following activities: display graphics, manage the objects that are displayed on the screen, compute object position and speed, keep track of scores, keep track of various stages of the game.

The Client–Server and other distributed architectural patterns

We discussed the Client–Server architectural pattern in detail in Chapter 3. Its basic principles are: a) there is at least one component that has the role of server, waiting for and then handling connections, and b) there is at least one component that has the role of client, initiating connections in order to obtain some service.

An important variant of the client–server architecture is the three-tier model under which a server communicates with both a client (usually through the Internet) and a database server (usually within an intranet, for security reasons). The server acts as a client when accessing the database server.

A further extension to the Client–Server architectural pattern is the Peer-to-Peer architectural pattern. A peer-to-peer system is composed of various software components that are distributed over several hosts. Each of these components can be both a server and a client. Any two components can set up a communication channel to exchange information as required. Figure 9.9

shows a peer-to-peer architecture for instant messaging. Messages no longer have to be sent through a central server. However, before two peers can communicate, they have to discover each other's existence. A central server may still be used for this.

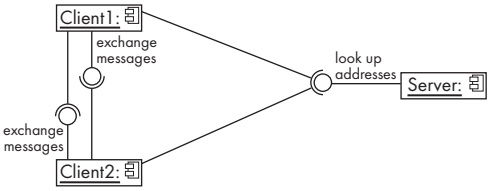

Figure 9.9 **A peer-to-peer architecture for instant messaging, retaining a central server only to look up the addresses of clients**

Distributed architectures help you adhere to design principles such as the following:

1. *Divide and conquer*. Dividing the system into client and server processes is a very strong way to divide the system. Each can be separately developed.
2. *Increase cohesion*. The server can provide a cohesive service to clients. For example, it can provide a single service with no side effects and therefore be functionally cohesive. A server also acts as a lower-level layer that the client accesses.
3. *Reduce coupling*. There is usually only one communication channel between distributed components, and the data being passed is usually simple messages. This helps reduce coupling, although there is normally control coupling (commands sent by the client control what the server does).
4. *Increase abstraction*. Separate distributed components are often good abstractions. For example, you do not need to understand the details of how a server operates.
6. *Increase reuse*. It is often possible to find suitable frameworks on which to build good distributed systems (e.g. OCSF). However, reusability may not be high since client–server systems are often very application specific.
7. *Design for flexibility*. Distributed systems can often be easily reconfigured by adding extra servers or clients. Furthermore, as discussed in Chapter 3, clients and servers can be developed by competing organizations, giving the customer a choice. However, changing the protocol of the system can be difficult.
9. *Design for portability*. You can write clients for new platforms without having to port the server.
10. *Design for testability*. You can test clients and servers independently.

⓫ *Design defensively.* You can put rigorous checks in the message handling code to ensure that no matter what messages you receive, your component will not crash.

Exercise

E181 The original Napster was the most famous and controversial of many peer-to-peer systems for sharing files; in its architecture, it retained a central registry of users. Other peer-to-peer file sharing systems, such as Gnutella and KAZAA, do not require a central registry. Do some research to determine how the various current file-sharing systems work, and how some are able to dispense with a central server. What are the advantages and disadvantages of the two approaches?

The Broker architectural pattern

The idea of the Broker architectural pattern is to distribute aspects of the software system *transparently* to different nodes. This is illustrated in Figure 9.10.

Figure 9.10 **The Broker architectural pattern**

Using the Broker architecture, an object can call methods of another object without knowing that this object is remotely located. The use of the Proxy design pattern (discussed in Chapter 6) can help achieve this goal. A Proxy object calls the broker, which determines where the remote object can be found.

CORBA is a well-known open standard that allows you to build this kind of architecture – it stands for Common Object Request Broker Architecture. Java has many classes that allow you to use CORBA facilities. There are also several other commercial architectures that also provide broker capabilities.

The Broker pattern is particularly useful in helping you follow these design principles:

❶ *Divide and conquer.* The remote objects can be independently designed.

❺ *Increase reusability.* It is often possible to design the remote objects so that other systems can use them too.

❻ *Increase reuse.* You may be able to reuse remote objects that others have created.

❼ *Design for flexibility.* The broker objects can be updated as required, or you can redirect the proxy to communicate with a different remote object.

❾ *Design for portability.* You can write clients for new platforms while still accessing brokers and remote objects on other platforms.

⓫ *Design defensively.* You can provide careful assertion checking in the remote objects.
Note that the separation of data between a proxy and a remote object tends to reduce communicational cohesion.

Exercise

E182 Do some research to determine the main broker platforms and their differences.

The Transaction Processing architectural pattern

In the Transaction Processing architectural pattern, a process reads a series of inputs one by one. Each input describes a *transaction* – a command that typically makes some change to the data stored by the system. There is a transaction dispatcher component that decides what to do with each transaction; this dispatches a procedure call or message to a component that will handle the transaction.

For example, in the airline system, transactions might be used to add a new flight, add a booking, change a booking or delete a booking. This is illustrated in Figure 9.11.

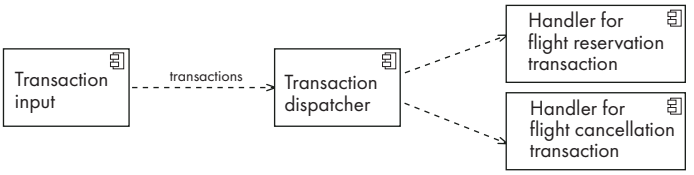

Figure 9.11 The transaction processing architecture used in an airline system

The dynamic binding and dispatching of polymorphic methods is a further example. In fact, it is best to design a system to use this implicit transaction-processing mechanism where possible, rather than designing an explicit version.

Transaction processing systems are often embedded in servers. A typical example is a database engine, where the transactions are various types of queries and updates. The command handler mechanisms in SimpleChat's server and client classes are, in fact, transaction dispatchers, although the pattern would be more strongly followed if they dispatched each command to a separate method.

Transactions themselves vary in their level of complexity. In many cases an update transaction requires that several separate changes be made to a database. For example, as we saw in Chapter 5, booking a passenger on a flight requires creating a `Booking` object, as well as creating bi-directional links from it to a `SpecificFlight` and a `PassengerRole`. It is essential that this whole transaction be fully completed, or else not done at all. Transaction dispatchers and handlers therefore work together to assure the *atomicity* of transactions.

Many transaction processing systems work in environments where several different threads or processes can attempt to perform transactions at once. In these environments the data to be modified by the transaction must be locked first, and the lock must be released afterwards. The situation is further complicated by the fact that an application may need to do a query transaction, then process the data, and finally do an update transaction on the data, while being assured that another thread has not changed the data in the meantime. If a lock is used, then other transactions are delayed until the whole process is complete. There are several strategies for handling this situation – you should read a database design book to learn about them. However, in the next chapter we will raise some relevant issues when we talk about testing systems that have multiple threads.

The Transaction Processing pattern is particularly useful in helping you follow these design principles:

1. *Divide and conquer*. The transaction handlers are suitable system divisions that you can give to separate software engineers.
2. *Increase cohesion*. Transaction handlers are naturally cohesive units. They may exhibit functional, sequential or procedural cohesion. However, they tend not to exhibit communicational cohesion.
3. *Reduce coupling*. Separating the dispatcher from the handlers tends to reduce coupling. However, you have to be careful that the coupling among the transaction handlers is kept under control.
7. *Design for flexibility*. You can readily add new transaction handlers.
11. *Design defensively*. You can add assertion checking in each transaction handler and/or in the dispatcher.

The Pipe-and-Filter architectural pattern

The Pipe-and-Filter architectural pattern is also often called the *transformational* architectural pattern. It works as follows. A stream of data, in a relatively simple format, is passed through a series of processes, each of which transforms it in some way. The series of processes is called a *pipeline*. Data is constantly fed into the pipeline; the processes work concurrently (conceptually at least) so that data is also constantly emerging from the pipeline.

The strength of this pattern is that the system can be modified easily by adding or changing the transformational processes. This is easiest if, at most stages of the pipeline, the data has the same general form. For example, the data might be simply a stream of characters. The processes might do such things as converting the characters to upper case; removing unneeded characters; or encrypting the text.

Another example of a pipe-and-filter architecture is a speech transmission system. This would continuously read sound coming from microphones, process it in various ways, compress it, transmit it over a network and then

regenerate sound at a remote location. It would use several different transformational components to do this. The architecture is illustrated in Figure 9.12.

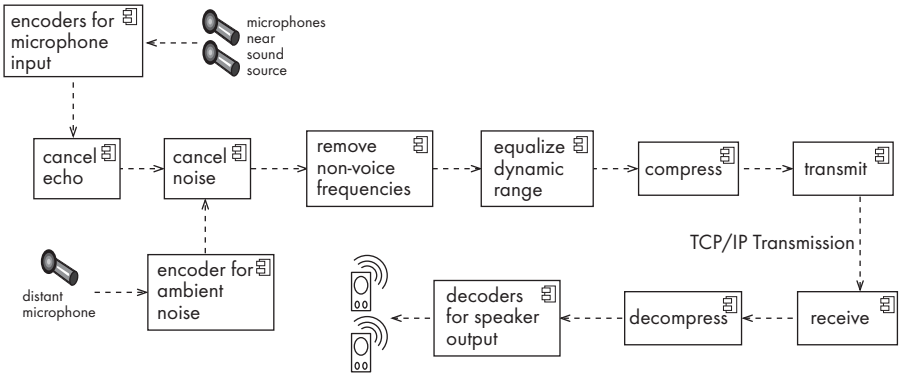

Figure 9.12 **A pipe-and-filter architecture for sound processing**

Some of the transformational processes are true filters: 'cancel echo', 'cancel noise', 'remove non-voice frequencies' and 'equalize dynamic range' simply remove some of the information, but leave it in the same format. The other processes convert the information into distinctly different formats.

The 'cancel noise' process illustrates how two pipelines can join together. It is also possible for a pipeline to split into two.

In Chapter 3, we saw the use of the classes `ObjectOutputStream`, `DataOutputStream`, `OutputStream`, and the corresponding input streams. These act like filters passing data to each other: for example `ObjectOutputStream` will convert an arbitrary object into bytes which it then passes to `OutputStream` for transmission; after transmission, an `InputStream` will receive the data and pass it to an `ObjectInputStream` for reconstruction of the original objects.

A pipe-and-filter system provides fulfils the following principles:

1. *Divide and conquer*. The separate processes can be independently designed.
2. *Increase cohesion*. The processes generally have functional cohesion.
3. *Reduce coupling*. The processes have only one input and one output, normally using a standard format, therefore coupling is very low. Type use coupling can become an issue if the format of the data needs to change.
4. *Increase abstraction*. The pipeline components are often good abstractions, hiding their internal details.
5. *Increase reusability*. The processes can often be used in many different contexts.
6. *Increase reuse*. It is often possible to find reusable components to insert into a pipeline.

7. *Design for flexibility.* There are several ways in which the system illustrated in Figure 9.12 is flexible:

   - Almost all the components could be removed. For example, 'cancel echo' or 'cancel noise' could be removed. In fact the encoder for microphone input could be directly connected to the decoder for speaker output, drastically shortening the pipeline.
   - Components could be replaced with different implementations. For example the 'transmit' and 'receive' components could be replaced in order to allow communication over a different type of network. The 'compress' and 'decompress' components could also be replaced in order to use different algorithms.
   - New components could be inserted, for example to perform encryption.
   - Certain components could be reordered. For example, removing non-voice frequencies could be done before canceling noise.
   - The encoders for microphone input and for ambient noise could be instances of the same component type.

10. *Design for testability.* It is normally easy to test the individual processes.
11. *Design defensively.* You can rigorously check the inputs of each component, or else you can use design by contract, writing careful preconditions and postconditions for each component.

Exercise

E183 Outline the design of a pipe-and-filter system to analyze a continuous stream of quotations from the stock market. One filter might extract only those stocks in which you are interested. Another process might add information to each stock quote, such as the total number of stocks you own, and the amount of profit or loss you have made on the stock. Some components might analyze quotes traveling through the pipe, and might alert you about whether you should buy or sell the stock.

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

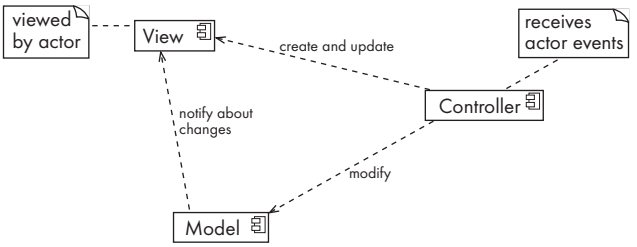

Figure 9.13 The Model–View–Controller (MVC) architectural pattern for user interfaces

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.

Exercise

E184 Imagine you are designing a system where a user can select a given company and obtain the price of its stock; the stock price is refreshed at regular intervals. Identify the model, the view and the controller in this application. Draw a high-level sequence diagram of a typical interaction.

The Service-Oriented architectural pattern

Now that the Internet has become so pervasive, a new architectural pattern has become prominent: *Service-oriented architecture*. This architecture organizes an application as a collection of services that communicate with each other through well-defined interfaces. In the context of the Internet, the services in question are called *Web services*.

A Web service is an application accessible through the Internet that can be integrated with other web services to form a *Web-based application*. To use a web service, you send a correctly formatted http request to an http server. Unlike normal http requests this is done 'behind the scenes' by the Web application program, not a browser being used by an end user. The server will run the service application and return the response as a document, typically structured using a language called *XML*. This is illustrated in Figure 9.14.

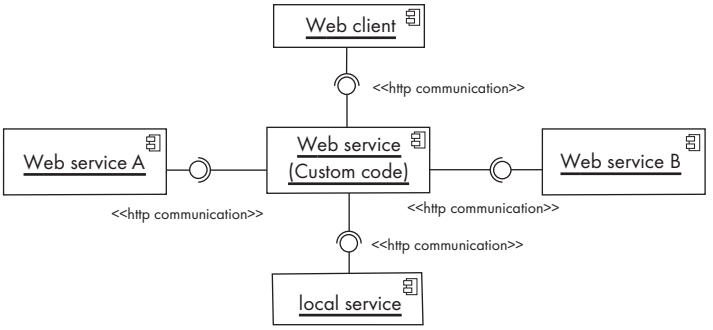

Figure 9.14 **The Service-oriented architectural pattern**

In Figure 9.14, a client program we are developing obtains information from a Web service provided by some company on the Internet. This in turn calls upon two other Web services, A and B, also available on the Internet. In addition, it uses Web-services protocols to access a local server operated by the same company.

A key aspect of the Web service architecture is that its different components communicate with each other using open Web standards. You build applications whose components are bound together over the Internet. The result is that a Web service can be accessed by many applications in many locations.

Web services can perform a wide range of tasks. Some may handle simple requests for information while others can perform more complex business processing. Enterprises can use Web services to automate and improve their operations. For example, an electronic commerce application can make use of web services to:

- access the product databases of several suppliers;
- process credit cards using a Web service offered by a bank;
- arrange for delivery using a Web service offered by a shipping company.

The toughest challenge facing the developer of a Web service is *security*. The service opens the enterprise's business operations and data to remotely distributed clients. Special care must therefore be taken to protect both customer and business information. Other important considerations are reliability, availability and scalability of the Web service.

Two examples of well-known platforms you can use to build web-services-based applications are Sun's J2EE platform and Microsoft's .NET Framework. These can be seen as large horizontal frameworks providing, among many other things, the required functionality to build interoperable services. They both provide flexible security models and offer other mechanisms to assist with scalability and reliability.

The Web services pattern helps you to adhere to the following design principles:

1. *Divide and conquer*. The application is made of independently designed Web services, which are distributed and accessible through the Internet.
2. *Increase cohesion*. Normally, the Web services are structured as layers, where high-level services are built on top of lower-level services. Some Web services can also exhibit functional cohesion.
3. *Reduce coupling*. Web-based applications are loosely-coupled applications built by binding together distributed components.
4. *Increase abstraction*. Since the clients communicate with each Web service through well-defined interface, no details about the particular implementation of a given Web service need to be known. However, communication with a Web service can sometimes be quite complex if the protocol is not defined at the right level of abstraction.
5. *Increase reusability*. A Web service is a highly reusable component.
6. *Increase reuse*. Web-based applications are built by reusing existing Web services.
8. *Anticipate obsolescence*. If the technology used inside a given Web service becomes obsolete, then a new implementation of this service can be offered without impacting the applications that use it. The web-service architecture can be seen as a client–server architecture where the obsolescence risk is reduced by relying on open communication standards.
9. *Design for portability*. A service can be implemented on any platform that supports the required standard protocols.
10. *Design for testability*. Each service can be tested independently.
11. *Design defensively*. Web services enforce defensive design, since many different applications, written by various developers (including malicious ones), can access the service. Each Web service has therefore no choice but to detect and reject any inappropriate or improperly formatted request for service.

Exercise

E185 Search the web for a company offering a public Web service. Describe the service offered and give an example of an application that could use it.

The Message-Oriented architectural pattern

Also known as Message-Oriented Middleware (MOM), this architecture is based on the idea that since humans can communicate and collaborate to accomplish some task by exchanging emails or instant messages, then software applications should also be able to operate in a similar manner. The core of the architecture is an application-to-application messaging system. Senders and receivers need only to know what are the message formats; that is, a receiving (or sending) application does not have to know anything about the software component that sent (or received) the message. Moreover, the two communicating applications do not even have to be available at the same time. This is illustrated in Figure 9.15.

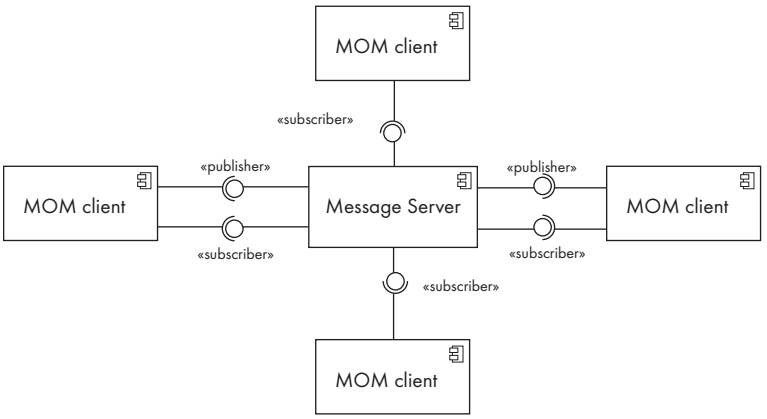

Figure 9.15 The Message-Oriented architectural pattern

The message, which is central to MOM architectures, is a self-contained package containing application data plus some network routing headers used by the messaging system to transmit the message across the network. The messages are sent through virtual channels, also called *topics*. Software components that send messages to topics are called *publishers*. To receive messages, an application must *subscribe* to a topic. In general, any message sent to a given topic is delivered to all the topic's subscribers, each of them receiving a copy of the message. This has similarities to the Observer design pattern discussed in Chapter 6.

The application can choose to ignore a received message or react to it by, for instance, sending a reply containing requested information. The exchange of these messages is governed by two important principles. First, message delivery is completely asynchronous; that means the exact moment at which a given

message is delivered to a given subscribing application is unknown. Secondly, reliability mechanisms are in place such that the messaging system can offer the guarantee that a given message is delivered once and only once.

Text messaging using cellular phones can be seen as a simple message-oriented application, but more complex systems can also adopt this architecture. Consider, for example, a sales company where various vendors sell products to consumers. When an item is sold, the vendor's software system can automatically send a message to the appropriate topic. The inventory component, one of the subscribers of this topic, would then be informed and initiate the product delivery process. At the same time, the sales office will receive the same message and use it to, let's say, accumulate statistics about sales trends. The accounting department would calculate the vendor's commission from the information received. And finally, another subscriber could be the factory that would adjust the production rate based on the number of products sold.

Clearly, the effectiveness of such system depends on the messaging system that is used to deliver the messages. Two approaches can be taken when designing such a system. The first is to use a centralized architecture where all messages transit through a message server that is responsible for delivering messages to the subscribers. This is the simpler model, but all functionality then relies on the server. The alternative is to use a decentralized architecture where message routing is delegated to the network layer and where some of the server functionality is distributed among all message clients. The *Java Message Service* (JMS) is an API providing all the required features to develop and deploy message-oriented applications. JMS-based components are portable and can be used by different applications that communicate using different messaging system service providers.

This pattern allows us to adhere to the following design principles:

1. *Divide and conquer*. The application is made of isolated software components, independently designed and distributed across a network.

3. *Reduce coupling*. The components are loosely coupled since they share only data format (a form of data coupling).

4. *Increase abstraction*. The quality of the abstraction mainly resides in the prescribed format for the messages. These are generally fairly simple to manipulate, all the application details being hidden behind the messaging system.

5. *Increase reusability*. A component will be reusable if its underlying prescribed message formats are flexible enough to adapt to different contexts.

6. *Increase reuse*. The different components can be reused as long as the new system can adhere to the formats proposed by each reused component.

7. *Design for flexibility.* It is always easy to update or enhance the functionality of a message-oriented system just by adding, removing or replacing components in the system.

10. *Design for testability.* Each component can be tested independently.

11. *Design defensively.* Defensive design in the context of message-oriented systems consists of validating all received messages before processing them and ignore the ones that cannot be properly interpreted.

Exercise

E186 Draw a diagram showing the components of an online auction application designed using a message-oriented architecture. Describe the messages to be exchanged.

E187 Sketch a possible architecture for the following systems. It will be helpful first to sketch the most important use cases. Describe which architectural patterns you are using.

(a) A corporate payroll system. All the information about employees, including their monthly salary and bank account, is kept in a database. Every two weeks, the system pays employees by depositing their salary in their bank account.

(b) A system to buy stocks on the Internet. Clients perform their transactions using a web browser. They are to access their accounts by providing a user ID and password. When an order is made, the system must process it by communicating with the appropriate stock market. The system also has to interact with the bank account of the client once the amount of the transaction is determined.

(c) The GANA system whose requirements were described in Chapter 4.

(d) A system for analyzing signals from a radio-telescope to see if there are signs of extra-terrestrial intelligence. This system takes tapes containing radio signal data, and divides them up into small 'work units' that are distributed to hundreds of thousands of computers running screen savers. These screen savers filter and transform the data in various ways, and then analyze it to detect different kinds of signals. Results are returned to the central site for further analysis. See setiathome.ssl.berkeley.edu for a real implementation of such a system.

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.

9.9 Difficulties and risks in design

- **Like modeling, design is a skill that requires considerable experience**. Design requires weighing many alternatives; however, to do so requires knowing about the alternatives and also being able to evaluate their consequences.
  Resolution. Do not attempt to design large systems until you have experienced a wide variety of software development projects. Actively study designs of other systems, including designs that have been found to be both good and bad.
- **Poor designs can lead to expensive maintenance**. A system with high coupling and low cohesion will be very difficult to change, will have many more defects and will deteriorate more rapidly than a well-designed system.
  Resolution. Follow the principles discussed in this chapter. Also, use the modeling techniques and patterns discussed in this book. Hold design reviews so that others can find flaws in your designs.

- **Ensuring that a software system's design remains good throughout its life requires constant effort.** Design tends to deteriorate as a result of adding new features that were not anticipated in the original design. The design also deteriorates as a result of people not understanding it when they make modifications.
  Resolution. Make the original design as flexible as possible so as to anticipate changes and extensions. Ensure that the design documentation is usable and at the correct level of detail so that maintainers will be able to make effective use of it. Ensure that change is carefully managed, with all changes to the requirements and design undergoing review. If it appears that implementing a new requirement would necessitate undermining the integrity of the architecture, then perform re-engineering to change and improve the architecture first.

9.10 Summary

In this chapter we have discussed principles, techniques and patterns that, if applied carefully, should lead to software that is flexible and maintainable.

Four of the most important principles are: divide up the system so that you can better handle its complexity, increase cohesion so that each component has a clearly identifiable purpose, reduce coupling so that there is minimal dependency between components, and increase abstraction so that you can work with the design without having to understand its details.

It is important to try to build components that have functional cohesion (the component computes just one output with no side effects), layer cohesion (the component provides a unified set of services at a certain level of abstraction) or communicational cohesion (the component manages all the interaction with a particular class of data). It is also important to reduce content coupling (surreptitious modification of data that is internal to a component), common coupling (use of global data) and control coupling (one component completely controls what another component does).

Other important principles are: strive for high flexibility, reusability, reuse, portability and testability. You should also anticipate obsolescence and design defensively.

Techniques for making good design decisions include carefully analyzing each option against the objectives and priorities, as well as performing cost–benefit analysis to determine whether a particular option is worthwhile.

Software architecture is a cornerstone of any software engineering project. The architectural model forms the basis on which the rest of the system is built. It allows you to divide the system into subsystems, and to distribute the work to team members. It also allows those team members to communicate effectively, and to plan for eventual extension of the system.

The architecture describes the subsystems, the interfaces between those subsystems, as well as their dynamic behavior, interactions and shared data. The architectural model can use all the various kinds of UML diagrams, including

the package diagrams, deployment diagrams and component diagrams which we introduced in this chapter.

When building an architecture, you should try to compose the system from various well-known architectural patterns. These include the Multi-Layer pattern, the Client–Server and other distributed patterns, the Pipe-and-Filter pattern, and the Broker pattern.

All aspects of design should be communicated to readers using carefully written documentation. The readers will include those who are implementing the design, those who will modify it and those who will develop other systems that interface with it. Remember to include rationale in your documentation.

9.11 For more information

- J. Bosch, *Design and Use of Software Architectures*, Addison-Wesley, 2000
- M. Jazayeri, A. Ran, F. Van Der Linden and Philip Van Der Linden, *Software Architecture for Product Families: Principles and Practice*, 2nd edition, Addison-Wesley, 2003
- L. Bass, P. Clements, R. Kazman and K. Bass, *Software Architecture in Practice*, Addison-Wesley, 1998
- C. Hofmeister, Robert Nord and Dilip Soni, *Applied Software Architecture*, Addison-Wesley, 1999
- D. Smith, *Designing Maintainable Software*, Springer Verlag, 1999
- A. Rollings and Dave Morris, *Game Architecture and Design: A New Edition*, Coriolis Group, 2003
- M. Fowler, *UML Distilled*, 3rd edition, Addison-Wesley, 2003
- R. Monson-Haefel, D. Chappell, *Java Message Service*, O'Reilly, 2000
- Software architecture resources on the web: http://www.serc.nl/people/florijn/interests/arch.html
- Bredemeyer's resources for software architects: http://www.bredemeyer.com
- CORBA: http://www.corba.org
- The Microsoft web services developer center: http://msdn.microsoft.com/webservices/
- The Sun web site on the J2EE technology: http://java.sun.com/j2ee/

Project exercises

E188 Create requirements documents for the following advanced features of the SimpleChat system.

(a) A capability to allow several servers to be linked together. A user could connect his or her client to any of the servers. The functions of the system would appear to the end-users to work in the same way as if there was just one server.

(b) A capability to allow users to send files to each other. The receiving user would have to acknowledge that he or she is willing to accept the file.

(c) A 'buddy-list' capability. A client can ask to be notified when any of a particular subset of other clients is logged on (currently the system tells you the complete set of people who are logged on).

(d) The ability to conduct votes among all the people on a channel. One user, who proposes a vote, specifies the question and several alternative answers that people can vote for. Voting can be done in a way that is visible to everybody, or else secretly (at the discretion of the proposer). Whenever the proposer desires, the results can be tabulated and transmitted to everybody.

E189 Create a design for each of the features you specified in the last exercise. If you discover defects in the requirements, then update them.

E190 Implement the features you designed in the last exercise. If you discover flaws in the requirements or design, then update those documents as needed.

E191 This exercise requires you to do some research into basic cryptographic techniques. Specify and design an extension to SimpleChat that would allow you to encrypt all the communication that occurs in a particular channel.

E192 Create a complete design document, including an architectural model, for the Small Hotel Reservation System.