3 Basing software development on reusable technology

In the last chapter, we refreshed your knowledge of the object-oriented paradigm, an important software development technology that can be used to construct complex software systems. It would be nice, however, if instead of developing an entire system from scratch, you could simply adapt an existing system to meet your needs. In other words, *reuse* is one of the keys to successful software development. We will start our exploration of the software development process by looking at a technology called *frameworks* that promotes reuse.

In this chapter we will also introduce the client–server architecture, one of the most widely used ways of structuring software systems. We will then introduce a framework specifically designed for this book that allows software developers to rapidly build many different client–server systems.

In this chapter you will learn about the following

- Frameworks, reusable software subsystems that implement important facilities which many applications can use.
- The client–server architecture, an important way of designing programs in which the software is divided into two main parts: a client program which runs on each user's computer, and a server program with which each user's client communicates in order to obtain services.
- A client–server framework written in Java. We will use this as the basis for many of the exercises presented in the book.

3.1 Reuse: building on the work and experience of others

Where feasible, software engineers should avoid re-developing software that others have already developed; in other words, they should try to *reuse* others' work.

In order to facilitate reuse, software engineers should also make their designs *reusable*. This means designing and documenting software so that it is understandable and flexible enough be used in a variety of different systems.

The following are some of the types of reuse practiced by software engineers, in increasing order according to the potential amount of work that can be saved by the reuse:

- **Reuse of expertise**. Software engineers who have many years of experience working on projects can often save considerable time when it comes to developing new systems because they do not need to re-think many issues: their past experience tells them what needs to be done. If such people write articles describing their experiences, this can help others to do better engineering work.

- **Reuse of standard designs and algorithms**. There are thousands of algorithms and other aspects of designs described in various books, standards documents and articles. These represent a tremendous wealth for the software designer, since all he or she needs to do is to implement them if they are appropriate to the current task.

- **Reuse of libraries of classes or procedures, or of powerful commands built into languages and operating systems**. Libraries and commands represent implemented algorithms, data structures and other facilities. Software developers always do this kind of reuse to some extent since all programming languages come with some basic libraries. The more powerful the facilities that come with a programming language, the more powerful and 'high level' the language is. Applications like spreadsheets, word processors and database programs have built-in languages with commands for such things as sorting, searching and displaying dialogs. Using these languages, which are often called *fourth-generation languages*, is an important form of reuse.

- **Reuse of frameworks**. Frameworks are libraries containing the structure of entire applications or subsystems. To complete the application or subsystem, you merely need to fill in certain missing details. A framework can be written in any programming language and can vary considerably in sophistication and detail. We will discuss them in more detail in Section 3.3.

- **Reuse of complete applications**. You can take complete applications and add a small amount of extra software that makes the applications behave in special ways the client wants. For example, you might take a standard email application and add a feature that would always update its 'address book' with data from the company's employee and client databases. This type of reuse is often called reuse of commercial off-the-shelf or *COTS* software, and the extra code written is often called *glue* code. It is common to write the glue code using *scripting* languages which run using an interpreter.

Newton on reuse
Reuse is not a new concept. It was Isaac Newton who said, 'If I have seen further it is by standing on the shoulders of giants.'

The elements reused in the latter three types of reuse are often collectively called *components*.

Unfortunately, reuse is not as extensive in software engineering projects as might be desirable. Some of the reasons for this are outlined in the next section. In this book, we want to encourage you always to think in terms of reuse when you develop software. Therefore, as a major part of this chapter, we will present a reusable framework that will form the basis for many examples and exercises.

Exercises

E37 Search the Internet in order to build a list of sources of information about the following things which can be reused during software development. Rate each source on a scale from low to high, where low means the source is very uninformative (perhaps just offering to sell a product), and high means it provides a wealth of practical information.

(a) Wisdom and experience about software design (e.g. tips, guidelines etc.).

(b) Written descriptions of standard algorithms.

(c) Class libraries.

(d) Code repositories.

(e) Fourth-generation languages.

(f) Macro packages you can add to spreadsheet or word processor programs.

(g) Frameworks.

(h) Scripting languages used to glue together COTS programs.

E38 Pick a couple of the best sources of information from the last exercise and discuss how they can help you achieve the reuse objective.

3.2 Incorporating reusability and reuse into software engineering

In order for reuse to occur, software developers must not only reuse existing good-quality components, but must also contribute to reusable components that others can use.

Encouraging reuse: breaking the vicious cycle

Reuse and design for reusability, especially of frameworks, need to be made part of the *culture* of software development organizations. In the many organizations that do *not* practice reuse, software engineers tend to start design from scratch for each new application either because there are no reusable components available to reuse, or because they do not feel confident about reusing whatever is available.

Developers are often willing to reuse packages of code delivered with a programming language, but are reluctant to develop new ones, and are especially reluctant to develop entirely new frameworks.

There are several reasons for this reluctance:

- Developing anything reusable is seen as not directly benefiting the current customer – after all, the current customer only needs *one* application, so why take the extra time needed to develop something that will benefit *other* applications? This argument often seems particularly convincing when developers are under extreme deadlines.
- If a developer has painstakingly developed a high-quality reusable component, but management only rewards the efforts of people who create the more visible 'final product', then that developer will be reluctant to spend time on reusable components in the future.
- Efforts at creating reusable software are often done in a hurry and without enough attention to quality. People thus lose confidence in the resulting components, and in the concepts of reuse and reusability.

Therefore many organizations suffer from a vicious circle: developers do not develop high-quality reusable components, therefore there is nothing good enough to reuse. Since there is nothing good enough to reuse, software developers take so much time to develop applications that they lack time to invest in reusable frameworks or libraries.

This cycle can only be broken if software engineers and their managers recognize the following points:

- The vicious cycle exists, and costs money.
- In order to save money in the longer term, some investment in reusable code is normally justified.
- Developers should be explicitly rewarded for developing reusable components.
- Attention to quality of reusable components is essential so that potential reusers have confidence in them.
- Developing reusable components will normally simplify the resulting design, independently of whether reuse actually occurs.
- Developing and reusing reusable components improves reliability, and can foster a sense of confidence in the resulting system.

The latter three points are worth further discussion.

The quality of a software product is only as good as its lowest-quality reusable component. It is no wonder then that many developers refuse to reuse components in which they lack confidence. To combat this, development of reusable components should be treated just like development of complete applications. You need to do proper domain and requirements analysis for the component; to design and document it properly; and to ensure its quality through testing and inspection. We will discuss these activities later in the book. In addition, it is important that software engineers be always available to properly maintain a reusable component. If all of the above are performed, then the component should be of high quality and hence it is more likely to be reused.

The process of developing reusable components, as part of a larger software project, can have significant benefits, even if the components are never reused outside the project. Looking at a problem at a more general level tends to make it easier to understand: details relevant to only certain specific cases are discarded, which leads to better abstractions and a simpler structure of the resulting design. Also, the very process of developing reusable components separately from their target system reduces the interconnections among parts of the system, a quality we will call low coupling, and discuss in detail in Chapter 9. This low coupling makes the resulting application easier to understand, modify and test.

In addition to simplifying design, reusable software tends to be more reliable. The more places the reusable components are used, the more testing they get. Also, they will be used in different contexts, thus their weak points are more likely to be exposed. When developing a new system, you can substantially increase confidence in it by composing it mostly of components that have already been thoroughly validated.

Making it possible to find reusable components

Even if reusable components are available, software engineers must be able to find them easily. An essential activity therefore is to carefully *catalog* and document all the reusable components.

This catalog must be easy to search and must be kept up to date. In particular, it is important to drop or *deprecate* older components that have been found to be unreliable or have been superseded by better components. Deprecating a component means declaring that it should not be used in subsequent designs, but remains available to support existing designs that incorporate it.

3.3 Frameworks: reusable subsystems

Developing and using frameworks is an excellent way to promote reuse and reusability.

Definition: a *framework* is reusable software that implements a generic solution to a generalized problem. It provides common facilities applicable to different application programs.

The key principle behind frameworks is as follows: applications that do different but related things tend to have similar designs – in particular, the patterns of interaction among the components tend to be very similar. This can be true even if the applications are in quite different domains. To develop a framework, you identify the common design elements and develop software that implements these design elements in a reusable way.

The key thing that distinguishes a framework from other kinds of software subsystem is that a framework is intrinsically *incomplete*. This means that there are certain classes or methods that are *used* by the framework, but which are missing.

The missing parts are often called *slots*. The application developer fills in these slots in an application-specific way to adapt the framework to his or her needs. The more slots that the application developer must fill, the more complex the framework is to use. At the same time, a framework with many slots tends to be more flexible and therefore you are more likely to be able to reuse it to create a wide range of applications across different domains.

Frameworks also usually have *hooks*: these are like slots, except that they are places where developers can add *optional* functionality of different kinds. We will see examples of hooks and slots in the framework we present later in this chapter.

Developers using frameworks not only fill slots and hooks, but they also use the services that the framework provides, i.e. methods that perform useful functions. The set of services, taken together, is often called the *Application Program Interface*, or *API*.

A framework enables the reuse of both design and code. The user of a framework not only reuses the overall design envisioned by the framework's designer, but also a body of code that implements that design.

The following are some examples of frameworks:

- **A framework for payroll management**. Most businesses have software that includes a payroll module. The rules and features needed in a payroll system will differ considerably, depending on the type of business, the local jurisdiction and other software the company uses. However, basic elements such as making regular payments, and computing taxes and other deductions, will always exist. Although it is possible to purchase complete payroll applications, many businesses are of sufficient complexity that such applications do not implement all the needed features and rules. Instead of developing a custom payroll package from scratch, several businesses could adapt a common framework to their individual needs.

- **A framework for a frequent buyer 'club'.** In order to encourage loyalty, many companies have a system that awards points to customers based on the amount they purchase. The details of such systems will differ from company to company, but they all have a lot in common. A company implementing a new frequent buyer club would do well to base it on a framework in order to avoid the cost of developing a system from scratch. An airline frequent-flier plan could be built using the same framework since it is merely a special kind of frequent buyer club.
- **A framework for course registration**. Each institution has its own academic rules, hence it is difficult to create a commercial application that can be bought off the shelf to automate student information systems. However, when software engineers are developing or replacing student information systems, they could benefit from basing their designs on a common framework.
- **A framework for e-commerce web sites**. Most e-commerce web sites are built on the same general model. There is a list of products to pick from; when an item is selected it is added to a shopping cart; the site then prompts for personal information and arranges for secure payment. Individual web sites will want to have special features to differentiate themselves in the market. However, developers could save a lot of work if they had a framework that implemented the above general model.

Frameworks and product lines

A *product line* (or *product family*) is a set of products built on a common base of technology. The various products in the product line have different features to satisfy different market requirements. Many consumer products are sold in product lines. For example, a company producing microwave ovens will likely produce a very basic model that they can sell cheaply, and successively more expensive models with increasingly sophisticated features.

The software industry is following the product-line model more and more. Underlying a software product line is a framework containing the software technology common to all the products in the line. Each product is then produced by varying the modules used to fill the hooks and slots; new product variations can be produced quickly and easily.

For example, the software controlling a line of microwave ovens will be based on a common framework. Each model in the line will then have different combinations of software and hardware features. Doing this is far more economical than designing each model separately.

Product lines are also found in many generic software products: you can often purchase stripped-down 'demo' or 'lite' versions of software, as well as 'pro' versions with extra features. Sets of software versions each tailored to specific languages or countries also represent product lines.

Horizontal and vertical frameworks

A framework can be *horizontal* or *vertical* (Figure 3.1). A horizontal framework provides general application facilities that a large number of applications can use. For example, if many applications need to have a 'preferences' dialog that allows users to specify many kinds of options, then a horizontal framework could be designed that would provide general 'preferences dialog facilities' for many different types of applications.

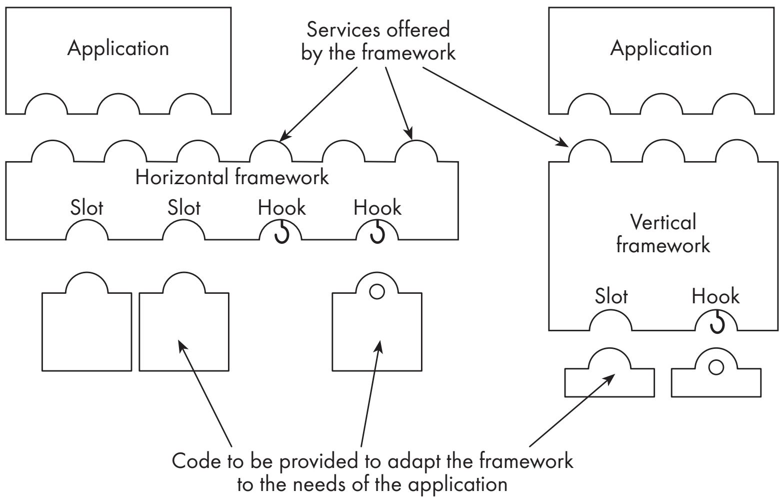
Application
Services offered by the framework
Application
Horizontal framework
Slot
Slot
Hook
Hook
Vertical framework
Slot
Hook
Code to be provided to adapt the framework to the needs of the application

Figure 3.1 **Horizontal and vertical frameworks showing services (at the top) and fillers of hooks and slots (at the bottom). One of the hooks is not filled**

A vertical framework, often also called an *application framework*, provides facilities that will allow easy development of a more specific class of application programs. The microwave oven, frequent-buyer and course registration frameworks are vertical in nature, while the e-commerce framework might be a hybrid – a vertical framework composed of several horizontal frameworks that perform the sub-functions (such as general secure payment facilities).

A vertical framework will have a more complete implementation, and may have fewer slots and hooks. An interface in Java can be considered an extreme example of a horizontal framework: there is no implementation, and all the specified methods represent slots that must be filled.

An application will typically use only a subset of the framework's services. For example, a framework for a rental store could do such things as manage membership, handle deposits, process rentals and returns, and compute penalties for late returns. A developer using this framework to build an application for a video rental store would likely ignore the facilities for handling deposits, but would take advantage of the membership facilities. When building a car-rental system, the opposite would be true.

In Section 3.6, we will be studying a framework for the development of client–server applications. This is a horizontal framework, since it is usable by a very large number of applications that require a client–server architecture, but does not itself provide any functions for the end-user.

Object-oriented frameworks

In the object-oriented paradigm, a framework is composed of a library of classes. The set of services – the API – offered by the framework is defined by the set of all *public methods* of the public classes.

Some of the classes in an object-oriented framework should be abstract. To use the framework in the context of a new application, the developer creates concrete classes that extend these abstract classes. The abstract methods in the abstract classes are the slots that are filled when concrete methods are created in the concrete subclasses.

Example 3.1 *Imagine you are designing a framework that different libraries (of books, not code) would be able to adapt to meet their needs. What kind of facilities would you want to provide if you were designing such a framework? In what ways do libraries differ such that they would need to use a framework rather than a complete application?*

Answer: common facilities a library framework might provide include:

- A user interface providing standard kinds of searches (e.g. by author, title and subject) and the ability to browse through lists of books and periodicals, or authors.
- Basic classes representing books, clients, loans etc., along with common operations that can be done with those classes.

Differentiating features of library systems might include:

- The cataloging scheme (e.g. Dewey Decimal or Library of Congress).
- The kind of information kept about each client and book (e.g. clients may have different privileges, such as to be able to borrow only certain types of books).
- Rules for types and lengths of loans, putting items on hold, payment of fines etc.
- The particular types of items that can be borrowed from the library. All libraries have books, but libraries may contain such specialized items as videos, maps or rare books that need special treatment.
- Specific data unique to this library such as a specific style of barcodes placed on books, multilingual support, etc.
- Specific hardware the library possesses, such as particular types of barcode scanners and checkout machines.

- The security mechanisms, such as who has authority to do what kind of operations. Login passwords in a university library might, for example, be integrated with login passwords for other university systems.
- Integration of the system with other systems such as online library resources, existing databases of books and periodicals, accounting systems (e.g. for fines).

Exercises

E39 Imagine you are designing a framework for the following classes of applications. Describe what services you might put in the framework. Answer this question using a simple list of things the system should be capable of doing.

(a) A *reservation* framework. This could be expanded into an application to reserve anything that needs reserving, e.g. dental appointments, meetings, tickets at the theater, etc.

(b) A *scheduling* framework. This could be expanded for scheduling meetings, trains, classes etc.

(c) A *language-processing* framework. This could be expanded to process a programming language, a database query language or a command language.

(d) An *editing* framework. This could be expanded to allow editing of text, spreadsheets, and elements of different kinds of diagrams. Think about common features of editing tools provide.

E40 For each of the three frameworks in the last exercise, what differentiating features would software developers need to provide to build specific applications? What hooks and slots should therefore be available?

E41 List as many types of applications as you can think of that might benefit from the development of the frameworks in Exercise E39, so as to reduce the work required to develop similar applications from scratch.

E42 Imagine an airline company asks you to develop the software for its frequent-flier program. You choose to attack the development of this system by first developing a framework. You consider two approaches:

(a) Developing and then adapting a *vertical* framework that provides the facilities needed by several types of frequent-flier programs.

(b) Developing and then adapting a more *horizontal* framework that encompasses any frequent-buyer program such as a hotel priority club, a book club or a video rental store membership club.

For each of these two approaches, sketch the resulting framework in terms of: (i) the services it would have to offer, (ii) the slots that should be present, and (iii) the hooks that would be useful.

E43 Prepare arguments for both sides of the following debate question: 'Resolved: when asked to develop a new frequent-flier system, developing a new frequent-flier framework would be a waste of time.'

3.4 The client–server architecture

Software architecture is the branch of software engineering that deals with how to organize and connect a set of software modules so that they can work together with each other. There are many well-known architectures – one of the most widely used is the client–server architecture. We will use this as the basis for much of the design work in this book; we will look at other architectures in Chapter 9.

We present the client–server architecture here since we want to introduce a client–server framework upon which we will build some example applications. You will find, in the coming sections, all the details you need to learn in order to understand how our client–server framework works. Once you understand this material, you will be able to reuse the framework to build a wide variety of applications.

A *distributed system* is a system in which computations are performed by separate programs, normally running on separate pieces of hardware, that co-operate to perform the task of the system as a whole. A *server* is a program that provides some service for other programs that connect to it using a communication channel. A *client* is a program that accesses a server. A client may access many servers to perform different functions, and a server may be accessed by many clients simultaneously. A *client–server system* is a distributed system involving at least one server and one client. A *peer-to-peer system* is a client–server system in which programs can act as both client and server for each other.

Figure 3.2 illustrates a server program communicating with two client programs. The vertical lines represent the three programs involved. After connecting, Client 1 sends a message, receives a reply and disconnects. Client 2 connects while Client 1 is still connected; it simply sends a message and then disconnects. This diagram is an example of a UML *sequence diagram*; we will study such diagrams in more detail in Chapter 8.

In general, the components of a client–server system interact as follows:

- The server starts running.
- The server waits for clients to connect. This waiting process is called *listening*.
- Clients start running and perform various operations, some of which require connecting to the server to request a service.

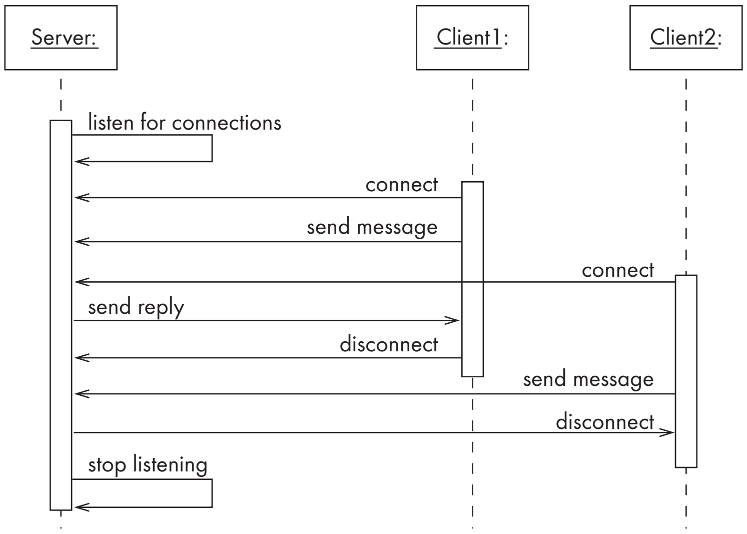

Figure 3.2 **A server program communicating with two client programs**

- When a client attempts to connect, the server *accepts* the connection if it is willing.
- The server waits for messages to arrive from connected clients.
- When a message from a client arrives, the server takes some action in response, then resumes waiting.
- Clients and servers continue functioning in this manner until one of them decides to shut down.

Normally, the action taken by the server includes sending a message back to the client. Most servers have to be able to handle connections from many clients and respond to messages from all the connected clients. How this is accomplished will be described below.

It is possible for the same program to be both a client and a server at the same time. For example, a database server might connect to another server in order to obtain additional data. It is also possible for the client and server to be located on the same computer, and run as separate processes. However, it is quite typical for them to be located on separate computers, perhaps in different geographical locations.

Table 3.1 lists some important kinds of systems that use the client–server architecture.

Comparing the client–server architecture to alternatives

You could also have some mechanism other than client–server communication for exchanging information. For example, one program could write a file, and another program could read the file, or else both programs could read and write

Table 3.1 Example client–server systems

System	*Clients*	*Server*
The World Wide Web	Browsers that display web pages and post forms, e.g. Netscape Navigator or Microsoft Internet Explorer	Web servers that manage sets of web pages (as well as CGI programs and servlets), and send information to browsers when sent a URL
Email	Programs that read and send email. For example, Microsoft Outlook, Eudora	A post-office program that receives email from remote sites and holds it until an email-reading client is activated. The program also forwards outgoing mail from the client to other sites
Network file system	Programs on any computer that access files that happen to be on other computers	A program whose main purpose is to allow clients on other computers to access files. Unix NFS and Novell NetWare are examples
Transaction processing system	Programs that send specific requests to perform some kind of transaction, such as debiting a bank account or booking an airline ticket	A program that centralizes all the functions of some business and processes transactions when they arrive
Remote display system	Programs that want to display information on the screen. Many Unix programs are capable of displaying graphical output on any computer running an X-Windows server	A program that manages the screen and allows applications, perhaps running on other computers, to display their output. A Unix X-Windows server is an important example
Communication system	A program that allows users to send a message or maintain a conversation with users on another computer	A program that routes messages. It can have features such as ‘forwarding’ that people are familiar with from the telephone network
Database system	Any application program that wants to query a database	A database management system that responds to requests to query or update the database

the same database. This could work for some kinds of communication, but would normally result in more complex and slower programs.

A single program that does everything can also be an alternative to a client–server system. However, the client–server architecture can have the following advantages:

- The computational work can be distributed among different machines. Designers can choose to centralize some computations on the server and distribute others to the clients. If everything is done on the server, then a powerful computer may be needed. On the other hand, if the clients take care of some computations then the server's workload will be lighter.
- The clients can access the server's functionality from a distance.
- The client and server can be designed separately, therefore they can both be simpler than a program that does everything. The development work can be done by independent groups, each only concerned with one part of the system (plus how the client and server communicate). Since the groups may be able to work on the client and server in parallel, they may be able to complete the whole system sooner.
- All the data can be kept centrally at the server, thus making it easier to assure its reliability. For example, it can be easier to ensure that regular backups are made of a single server's data, rather than trying to separately back up data saved by many separate programs.
- Conversely, distributing data among many different geographically distributed clients or servers can mean that if a disaster occurs in one place, the loss of data is minimized.
- The information can be accessed simultaneously by many users. It is possible to accomplish this using a single large program, but that approach tends to be more complex.
- Competing clients can be written to communicate with the same server, and vice versa; for example, different web browsers can communicate with the same web server. This can encourage innovation.

Exercises

E44 For each of the following systems, discuss under what circumstances it would be worth making it into a client–server system, as opposed to just creating a single program that does everything. In the case of a client–server system, indicate what work could be done by the server, and what by the client. In answering this question, make your best judgment, using whatever knowledge you already have about software applications.

(a) A word processor.

(b) A system for doctors to look up patient records when visiting a patient.

(c) A home alarm system that monitors various sensors such as motion detectors, smoke detectors and window-opening sensors.

E45 If you were designing a server for the following classes of applications, list the kinds of main activities that you might expect the server to do:

(a) A server for an airline reservation system.

(b) A server that contains the master list of toll-free telephone numbers that different telephone companies will need to access.

(c) A server that forms the center of a building alarm system; clients are individual controllers for devices around the building.

(d) Your favorite site for buying books on the Internet.

(e) A web-based course registration system.

Capabilities that must be provided when designing a server

A server has the following main activities to perform:

1. The server must **initialize** itself so that it is able to provide the required service. For example, a server that handles airline reservations might load data describing the available flights.
2. It must **start listening** for clients attempting to connect. Until it starts listening, any client that attempts to connect will not succeed.
3. It must handle the following types of *events* originating from clients, which can occur at any time:
   - ❑ It **accepts connections** from clients. This process will normally involve some form of validation to ensure that the client is allowed to connect. While a client is connected, the server keeps a record of the connection.
   - ❑ It **reacts to messages** from connected clients. This is the most important thing the server does. In an airline server a message could be a request to book a passenger, or a query to find out who is booked. In response to a message from a client, a server can do many types of things, including performing computations and obtaining information. Normally the server will send some information back to the requesting client; it might also send a message to another client or broadcast messages to many clients at once.
   - ❑ It **handles the disconnection** of clients. A client can request disconnection by sending a message to the server or by simply disconnecting itself; it might ‘disappear’ if it crashes, or if its network connection goes down; finally, the server might *force* a client to disconnect if the client is not ‘behaving’ well.
4. The server may be required to **stop listening**. This means that it will no longer

accept new client connections, but it will continue to serve the currently connected clients. This may happen when the number of connected clients becomes too high; in such a situation the server rejects new clients so that it does not run out of resources such as memory. When it has enough resources again, it can start listening again. The server may also choose to stop listening prior to shutting down, allowing the connected clients time to terminate their work.

5. It must cleanly **terminate**, i.e. shut down, when necessary. Shutting down cleanly means doing such things as notifying each client before terminating its connection.

The above main activities of a server are illustrated in Figure 3.3, which is an example of a UML state diagram. We will examine such diagrams in detail in Chapter 8; for now, we believe that the diagram is sufficiently self-explanatory.

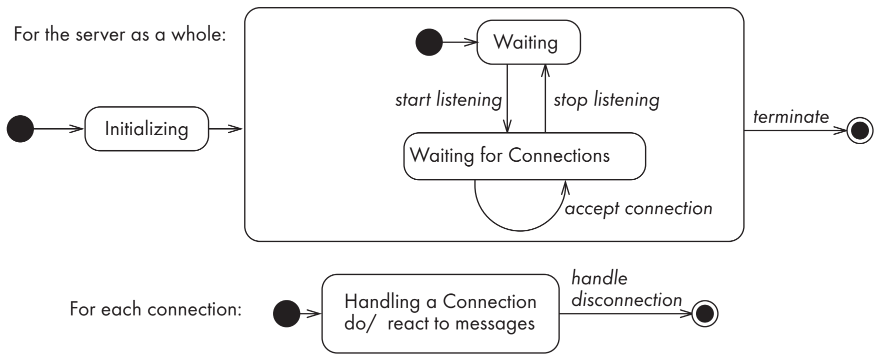

Figure 3.3 **The main activities performed by a typical server**

Later on in this chapter, we will see that in order to perform its work effectively, a server needs to use several concurrent threads.

Capabilities that must be provided when designing a client

A client has the following main activities to perform:

1. Like the server, a client must **initialize** itself so that it is able to communicate with the server. For example, it needs to know the network address of the server.
2. It performs some work, which includes:
   - ❑ Making a decision to **initiate a connection** to a server. If connecting to the server fails, or the server rejects the connection, the client may try again or may give up.
   - ❑ **Sending messages** to the server to request services.

3. It must handle the following types of events originating from the server, which can occur at any time:

   ❑ It **reacts to messages** coming from the server. Often, messages received from the server alternate with messages sent to the server – in other words, the messages from the server are replies to the client's requests. Sometimes, however, an unanticipated message might arrive from the server; for example, to announce that some new data is available or that the server is shutting down.

   ❑ It **handles the disconnection** of the server. This might occur because the server crashed or the network failed. It might also occur because either the client or server requested disconnection. The important issue is that the client knows it is no longer connected and makes decisions accordingly; one possible action is to attempt to reconnect.

4. It must cleanly **terminate**. This includes disconnecting from a server if it is still connected.

The above main activities of a client are illustrated in Figure 3.4, which shows one possible sequence of activities. Note that the 'regular' work of the client may need to proceed concurrently with the process of responding to events originating from the server. This is indicated in Figure 3.4 by the horizontal bars that show execution dividing into two distinct paths. We will consider concurrency in more depth in the next subsection.

Figure 3.4 is an example of a UML activity diagram; we will discuss these further in Chapter 8.

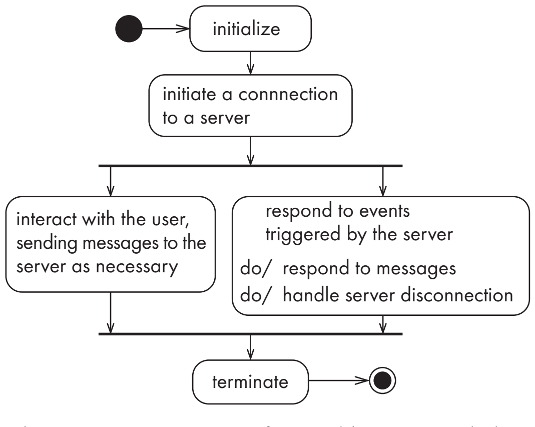
initialize
initiate a connnection to a server
interact with the user, sending messages to the server as necessary
respond to events triggered by the server
do/ respond to messages
do/ handle server disconnection
terminate

Figure 3.4 **The main activities performed by a typical client**

Concurrency in client–server systems

Client–server systems are inherently concurrent because the server runs at the same time as the clients, normally (but not necessarily) on different computers.

However, there is an added level of concurrency in both the client and server sides. As mentioned above, the client will normally be doing the following things concurrently:

- Waiting for interactions with the end-user, and responding when interactions occur.
- Waiting for messages coming from the server, and responding when messages arrive.

These generally have to be implemented using multiple threads of control that can be concurrently executed. Without this mechanism, when the client is waiting for one kind of input, it will not be able to respond to the other kind of input. An exception to this can occur in clients that do not need to interact with the user in any way.

Similarly, the server should normally have concurrent threads which do the following:

- Waiting for interactions with the user who is in charge of the server, and responding as necessary. As with the client, some servers can dispense with user interaction, but most will need a thread to handle basic controlling commands.
- Waiting for clients to try to connect and establishing connections as needed.
- For *each* connected client, waiting for messages coming from that client, and responding when messages arrive.

Servers thus normally operate with at least two concurrent threads, and in general n+2 threads where n is the number of connected clients. Figure 3.5 illustrates the various threads executing in a typical client–server system. In this diagram only one client (client A) is shown communicating with the server; however, a thread for a second dormant client (client B) is also shown.

Thin- versus fat-client systems

The work of a client–server system can be distributed in several different ways. In a *thin-client* system, the client is made as small as possible and most of the work is done in the server. In the opposite approach, called a *fat-client* system, as much work as possible is delegated to the clients. The two approaches are illustrated in Figure 3.6.

An important advantage of a thin-client system is that it is easy to download the client program over the network and to launch it. In Java, *applets* are usually thin clients because it is desirable for them to download rapidly. An advantage of fat-client systems is that since more computations are distributed to the clients, better use is made of available computational resources; the server can therefore be smaller or can be made to handle more clients.

One of the main considerations in choosing between a fat-client and a thin-client system is how intensively the system will use the network to communicate

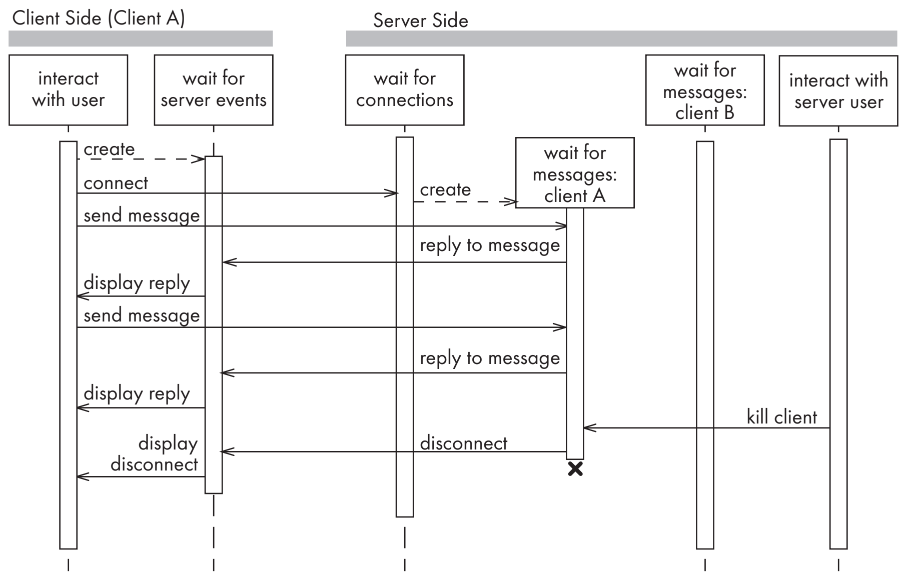

Figure 3.5 Threads in a client–server system

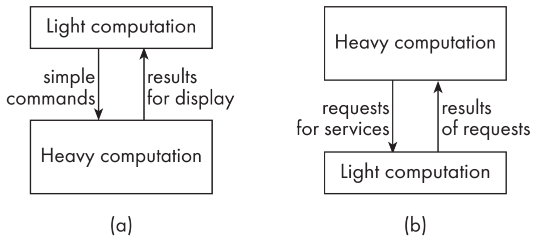

Figure 3.6 A thin-client system (a) and a fat-client system (b). The clients are at the top and the servers are at the bottom

– making the wrong choice can sometimes result in an overloaded network. Depending on the nature of the system, either a fat-client or a thin-client system may take the fewest network resources. In some cases, a thin-client system will need to communicate the least because it generally sends only simple user requests to the server. On the other hand, a thin client might need to communicate with the server much more frequently than a fat client and to download voluminous results of the server's calculations.

Exercise

E46 In each of the following systems, list: (i) the work normally performed on the server side; (ii) the work normally performed on the client side; (iii) the types of information transmitted in both directions over the network; (iv) whether the system is thin-client, fat-client or intermediate; (v) what could be done to increase or decrease the proportion of work done on the client side; (vi) what effects such changes would have on the network.

(a) to (e) The systems from Exercise E45.

(f)The world wide web in general (with browsers and web servers).

(g)The email system that you use.

Messages in a client–server system: communications protocols

The types of messages the client is allowed to send to the server form a *language*. The server has to be programmed to understand that language. Similarly, another language consists of the types of messages the server is allowed to send to the client.

When a client and a server are communicating, they are in effect having a conversation using these two languages. As with a human conversation, there have to be rules to ensure, for example, that the communicating parties take turns to speak. The rules also describe the sequences of messages that the client and server must exchange, in order to reach agreement on something or to accomplish some other task.

The two languages and the rules of the conversation, taken together, are called the *protocol*. The design of protocols can be very complex; in simple systems, such as those discussed in this book, the protocol is merely a list of service requests and their responses.

Example 3.2 *Sketch a protocol for a simple program for manipulating files on a remote computer.*

The following illustrates the kinds of messages sent between clients and the server.

Messages to server	*Possible replies to client*
`getFile` *name*	`fileContent`, `accessDenied`, `noSuchFileOrDir`, `failed`
`saveFile` *name content*	`successful`, `accessDenied`, `failed`
`rename` *oldname newname*	`successful`, `accessDenied`, `noSuchFileOrDir`, `failed`
`delete` *name*	`successful`, `accessDenied`, `noSuchFileOrDir`, `failed`
`listDir`	`fileList`, `accessDenied`, `failed`
`changeDir` *name*	`successful`, `accessDenied`, `noSuchFileOrDir`, `failed`
`createDir` *name*	`successful`, `accessDenied`, `failed`

The above protocol does not deal with such things as security and logging in; nor does it suggest how the information would be presented to the user in a friendly way.

Exercise

E47 Propose a simple protocol for the systems described in question E45.

Tasks of the software engineer when developing a client–server system

When designing a client–server system, the software engineer should make use of a framework that provides much of the underlying mechanism. We will describe such a framework later; however, the designer still has four key things to design:

1. The primary work to be performed by both client and server; i.e. the computations to be performed, data to be stored, etc.
2. How the work will be distributed – thin client, fat client, or intermediate.
3. The details of the set of messages that will be sent from the client to the server and vice versa in order to accomplish the main activities, i.e. the communications protocol.
4. What has to happen in the client and server when they start up, handle connections, send and receive messages, and terminate.

3.5 Technology needed to build client–server systems

In order to build a client–server system you need a computer network as well as software facilities for sending and receiving messages. There are several standards for data communication, and most modern programming languages include suitable data communication packages. This section discusses basic Internet and Java technology you can use to construct client–server systems.

Some important network concepts

In order to be able to understand how a client and a server communicate with each other, you must understand a few basic concepts about computer networks. Many books have been written about networks, but the few details discussed here will be enough to enable you to understand client–server design.

Since most computers today are connected to the Internet, we will assume that clients and servers will communicate with each other using the Internet's main communications mechanism, *TCP/IP*.

'IP' stands for 'Internet Protocol'. The main function of IP is to route messages from one computer to another. Long messages are normally split up into small pieces which are sent separately and then reassembled at the destination computer. Since the Internet is a large heterogeneous network of many computers and other devices, this routing process is quite complex. Luckily, Internet users rarely need to worry about the complexity.

How to find out the IP address and host name of a computer

Using a web browser, you can normally find out the IP address of your computer (and a lot of other information about your network connection) at privacy.net/analyze/.

In Windows XP you can find the IP address by first opening the 'Network Connections' control panel. Then click on the icon for a connection and look in the 'Details' tab. You can also issue the command 'ipconfig /all' to obtain very detailed information including your host name.

On Mac OS X, you can find out the IP address by looking at the 'TCP/IP' tab of the 'Network Preferences' panel.

On most varieties of Unix, including Mac OS X, or Linux, you can find out your host name by issuing the commands `hostname` or `uname -n`. You can normally look up the IP address corresponding to a host name using `ypcat hosts | grep <hostname>`. In addition, you can look up the host name corresponding to an IP address using the same command sequence.

On many computers (including both Windows and Mac) you can issue the command `netstat -a -p TCP` to determine which ports are in use.

'TCP' stands for 'Transmission Control Protocol'. TCP handles connections between two computers. A connection lasts for a period of time, during which the computers can exchange many IP messages. In addition to simply exchanging data, the computers use TCP to establish the connections, and to assure each other that the messages they have sent each other have been satisfactorily received. There is another mechanism called UDP that can be used instead of TCP; however, we will not discuss UDP here.

Each computer using IP is called a *host* and has a unique address. In IP Version 4, you may see this address written as four numbers (each from 0 to 255), separated by dots, such as 128.37.100.100; in IP version 6, to which many networks are moving, the address appears as eight hexadecimal numbers separated by colons. More commonly, however, you will see an IP address as a more human-understandable dot-separated series of words such as 'www.mcgraw-hill.com'. The numeric and word forms can be used interchangeably. The numeric form is normally called the *IP address*, while the word form is normally called the *host name*. If you strip off the first component of the host name (everything up to and including the first dot) then the remaining part typically represents a sub-network on which the host is running; this is often called a *domain*. In the above example, 'mcgraw-hill.com' is a domain. The '.com' is a *top-level domain*.

Several servers can run on the same host. Each server is identified by a *port* number, which is an integer from 0 to 65535. In order to initiate communication with a server, a client must know both the host name and the port number. By convention, port numbers from 0 to 1023 are reserved for use by specific types of servers; for example, web servers normally use port 80. Knowing this convention, a web browser that is only given a host name (in a URL) can connect to a web server by assuming that the server is at port 80. We therefore should not use port 80 for any other kind of server since confusion will result. In this book, we will by default run servers on port 5555 if it is not already occupied by some other server. In general, when you create a new server, you must pick a port number and publish both the host name and port number so that clients know where to connect. Taken together, the host name and port number are often just called the *address* of the server. By convention, if a client wants to talk to a server on the same computer, it can use the special host name *localhost* (IP address 127.0.0.1).

Establishing a connection in Java

Java includes a package specially designed to permit the creation of a TCP/IP connection between two applications: it is called `java.net`. The class `Socket` is the central element of this package; instances of this class encapsulate information concerning each connection. Both the client and the server must have an instance of `Socket` in order to exchange information.

Before a connection can be established, the server must start listening to one of the ports. To do this, it uses the resources of the class `ServerSocket`. This is typically done as follows:

```
ServerSocket serverSocket = new ServerSocket(port);
```

where `port` is the integer representing the port number on which the server should be listening.

In order for a client to connect to a server, it uses a statement like the following, passing the host name (or numeric IP address) and port number of the server:

```
Socket clientSocket = new Socket(host, port);
```

For the connection to be accepted, the server must have a thread constantly listening for connections using a statement like the following, embedded in a loop:

```
Socket clientSocket = serverSocket.accept();
```

The above statement will wait indefinitely in the `accept` method until a client tries to connect, then it will try to create an instance of `Socket` to handle the new connection. If this is successful, both client and server now have instances of `Socket` and can communicate freely with each other.

All of the above assumes the network is working properly, and appropriate values are specified for `host` and `port`. If communication fails for any reason, these statements will throw an `IOException`. Appropriate code must be written to handle such exceptions, e.g. notifying the user of the failure or trying again.

Once a connection is established, the exchange of communication may commence. From now on, both client and server can send messages to each other at any time. The connection is said to be *symmetric*, meaning that the client communicates with the server in the same way as the server communicates with the client.

Normally there will be two distinct *streams* of information: from server to client and from client to server. Each program uses an instance of `InputStream` to receive messages from the other program, and an instance of `OutputStream` to send messages to the other program. These classes are found in the package `java.io`, and their instances can be created as follows:

```
output = clientSocket.getOutputStream();
input = clientSocket.getInputStream();
```

When a message is sent from one program using its `OutputStream`, it may be read by the other connected program using its `InputStream`. However, `InputStream` and `OutputStream` deal with messages composed merely of bytes, the most primitive form of data. Programmers often want to exchange more sophisticated types of data without having to worry about how to translate them into a byte stream. To do this, Java provides a series of *filters* which convert the raw bytes into other forms. For example, `DataOutputStream` and `DataInputStream` allow direct transmission of the Java primitive types such as `int` and `double`. Another pair of filters, `ObjectOutputStream` and `ObjectInputStream`, allows the exchange of Java objects. For maximum flexibility, we will use this latter pair of classes in our client–server framework.

To send an object, Java uses a process called *serialization*. This is a technique by which every object is converted by an `ObjectOutputStream` into a binary form for transmission, and then reconstructed when it is received by an `ObjectInputStream`. Most objects can be serialized; the only requirements are that they be instances of classes that implement the interface `java.io.Serializable`, and that the data in their instance variables also be serializable. Serialization is also the mechanism used to save objects into a binary file.

In order to use an object stream, you must wrap it around a binary stream in the following manner:

```
output = new ObjectOutputStream(clientSocket.getOutputStream());
```

You can then send an object thus:

```
output.writeObject(msg);
```

In order to receive objects, you create an object input stream thus:

```
input = new ObjectInputStream(clientSocket.getInputStream());
```

and then arrange for the following statement to be executed in a loop:

```
msg = input.readObject();
```

The `readObject` method will wait until an object is received over the socket, or until an I/O error occurs. An I/O error will occur if the program at the other end of the connection is terminated.

3.6 The Object Client–Server Framework (OCSF)

In the next few sections we present a framework that can be used to develop any client–server system. We call this framework OCSF (Object Client–Server Framework) since it can be used to build a client–server system that exchanges Java objects. We will use the OCSF for the systems we develop in this book. In Chapter 6 we will extend the framework to make it more flexible.

You should attempt to understand completely how the OCSF functions. Not only will doing so ensure you understand the principles of frameworks and client–server systems in general, but it will also teach you about some of the subtleties of software design. Later in the book, some of the design issues raised here will be revisited.

To help you understand the framework, we provide a simple application in Section 3.9 that uses it. We also provide some project exercises where you change the application – modifying an existing application is one of the best ways to learn how it works.

The core of OCSF consists of three classes: one to implement the client and two to implement the server. The core classes are illustrated in Figure 3.7, along with their most important methods. The line with the asterisk connecting `AbstractServer` to `ConnectionToClient` indicates that there are many instances of `ConnectionToClient` associated with the server. The labels such as «control», «hook» and «slot» divide the methods into categories, which we will describe shortly.

In Chapter 6, we will discuss some additional classes that extend OCSF; there is no need to know anything about those to start working with OCSF.

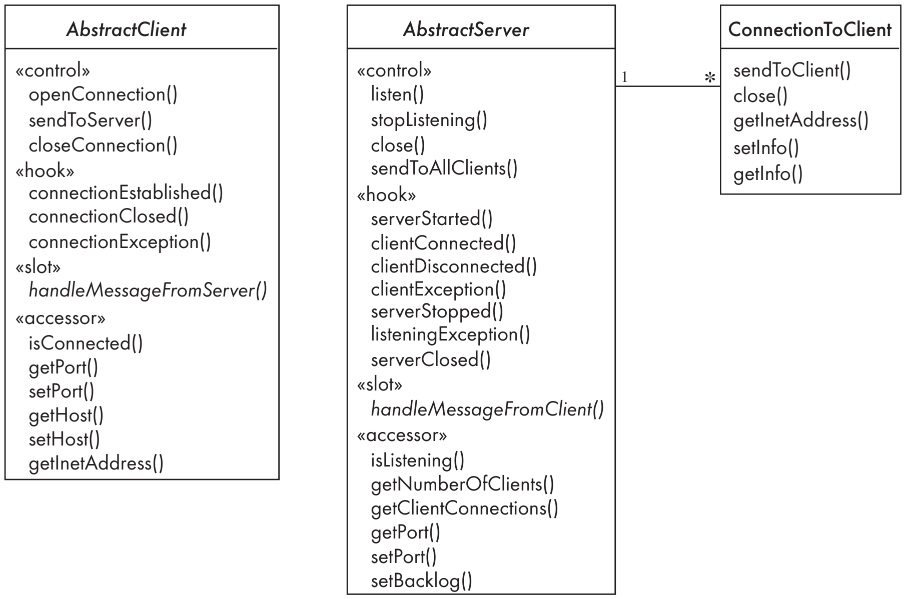

Figure 3.7 **The essentials of the core OCSF classes**

Programmers using OCSF never modify the framework's classes. Instead, a programmer should do the following to create an application:

- Create *subclasses* of the abstract classes in the framework.
- In these subclasses, write implementations of certain *slot* methods that are declared to be abstract in the framework classes.
- Also in the subclasses, override certain methods that are explicitly designed to be overridden. These are the *hooks* of the framework.
- In various parts of the application, call public methods that are provided by the framework. These public methods, which are the *services* of the framework, allow the application designer to control the client or server, and to find out information about them.

We will first discuss in detail the client side of the framework, and then the server side. In this chapter, your main objective should be to understand how to use the framework. To do that, you will have to understand how it works to some extent; however, you will probably obtain a more detailed understanding of that in later chapters.

Complete source code of the OCSF is found at www.lloseng.com.

3.7 Basic description of OCSF – client side

The client side of the OCSF consists of a single class `AbstractClient`. This is an abstract class that provides all of the facilities needed to connect and exchange objects with servers – with one exception: `AbstractClient` must be subclassed in order to implement the method `handleMessageFromServer` that takes appropriate action when a message is received from a server.

`AbstractClient` implements the `Runnable` interface. This is because we want the message waiting activity of its instance to run as a separate thread as described earlier. As an implementer of `Runnable`, `AbstractClient` has a `run` method which contains a loop that executes for the lifetime of the thread, receiving messages from the server and responding to them. We will discuss the internals of the `run` method a bit later.

The public interface of `AbstractClient`

The *public interface* to `AbstractClient` consists of the service methods that software developers who are using the class can access. In OCSF, as in other well-designed object-oriented systems, the public interface only provides a set of methods that can be called – it does not permit direct access to any variables.

The public interface of `AbstractClient` consists of three kinds of methods: a *constructor*, some methods that are used to *control* the client, and some methods used to *access* basic information about the client.

Public constructor. There is only one simple constructor in this class. It merely initializes variables representing the host and the port of the server to which the client will connect.

Public controlling methods. These methods provide services and do the bulk of the work of controlling the client. They are declared `final` so that they cannot be overridden by subclasses. The `final` declaration ensures that subclasses cannot create versions that contain bugs; however, it also means that subclasses cannot correct any design flaws in these methods. That puts a particularly strong responsibility for quality control into the hands of the framework's designers. The three key controlling methods are:

- `openConnection:` this connects, if it can, to a server at the host and port specified in the constructor (or subsequently using `setHost` and `setPort` described below). As soon as the connection to the server is established, this method starts the thread which will then run until the connection to the server is terminated.
- `sendToServer:` this sends a message to the server, if it can. The message can be any object.
- `closeConnection:` this stops the communication with the server and signals the thread to stop looping and hence terminate

All three of the above methods will throw an `IOException` if they fail – callers have to handle this in some way.

Utility accessing methods. These additional service methods are used to inquire about the state of the client or make minor changes to that state. They include:

- `isConnected:` allows callers to inquire whether the client is currently connected to a server.
- `getHostPort` and `getPort:` allow callers to inquire which host and port the client is connected to, or is prepared to connect to.
- `SetHost` and `SetPort:` allow callers to change the host and port of a disconnected client in preparation for the next call to `openConnection`.
- `getInetAddress:` provides some detailed information about the connection.

The callback methods of `AbstractClient`

In addition to the public interface, `AbstractClient` also contains several hook methods that are designed to be overridden by subclasses of the client, as well as one abstract slot method. The hooks and slots are called when particular events occur as the client operates. Methods like these are conventionally referred to as *callbacks*, since they are not called by the application code, but rather they represent *calls back* to the application code from methods in the framework.

Methods that may be overridden by subclasses (hooks). These may be overridden by subclasses and are called when various potentially 'interesting' events happen. If developers of subclasses of `AbstractClient` are interested in taking some action when these events occur, the developers can implement the methods. The default implementations do nothing.

- `connectionEstablished:` is called after a connection with a server is established.
- `connectionClosed:` is called whenever a connection with the server is terminated by the client.
- `connectionException:` is called when something goes wrong with the connection, such as when the connection is terminated by the server.

Method that *must* be defined in subclasses (slot). The only abstract slot method in `AbstractClient` is named `handleMessageFromServer`. This must be defined in subclasses and is called whenever a message is received from the server.

How an application developer should use `AbstractClient`

A developer who wants to design a client which uses the `AbstractClient` class need only do the following:

- Create a subclass of `AbstractClient`.
- In this subclass, implement the `handleMessageFromServer` slot method to do something useful with any messages coming from the server.
- Arrange for some code somewhere to create an instance of the new subclass of `AbstractClient` and to call `openConnection`.

In almost all clients, the developer will also want to do the following:

- Arrange for some code somewhere to send messages to the server using the `sendToServer` service method. It is possible to have a client that only receives messages from a server, and hence does not call `sendToServer`, but that would be rather unusual.
- Implement the `connectionClosed` callback to do something intelligent, such as notifying the user, when the connection to the server is terminated normally.
- Implement the `connnectionException` callback to deal with abnormal disconnection.

Not every application will need to use the other service methods, or override the other callback method (`connectionEstablished`).

A few details of the private internals of `AbstractClient`

Software developers do not, strictly speaking, need to know much more than the above to use `AbstractClient`. However, knowing a few details of how a class works can help a developer to diagnose problems and feel more comfortable using the class.

`AbstractClient` has the following instance variables:

- A `Socket`, `clientSocket`, which keeps all the information about the connection to the server.
- Two streams, an `ObjectOutputStream` (`output`) and an `ObjectInputStream` (`input`), that are used to transmit and receive objects using `clientSocket`.
- A `Thread`, `clientReader`, that runs using `AbstractClient`'s run method.
- A `boolean` variable, `readyToStop`, used to signal when the thread should stop executing.
- Two variables storing the `host` and `port` of the server.

The thread starts running when `openConnection` calls `start` which in turn calls `run`. The loop inside `run` repeatedly waits for a message to come from the server by calling the `readObject` method of the `ObjectInputStream`. When a message is received, the `run` method then responds by calling the application's implementation of `handleMessageFromServer`.

Complete source code for `AbstractClient` is found on the book's web site. You may find it useful to study the code, following the above explanation. We suggest you do the exercises at the end of the chapter to test your understanding.

3.8 Basic description of OCSF – server side

The server side of OCSF is slightly more complex than the client side since it has two classes, not one. The two classes are needed because, as discussed in Section 3.4, the server has to implement both the thread that listens for new connections (`AbstractServer`) and the threads that handle the connections to clients (`ConnectionToClient`).

The public interface of `AbstractServer`

As with `AbstractClient`, there is a limited set of public methods (the API) that provide all the services of this side of the framework.

The public constructor. `AbstractServer` has only one constructor, which takes a port number on which the server will listen. The port number can be changed later, if needed.

The public controlling methods. Similarly to the client side, the `AbstractServer` has a set of methods that can be used by subclasses to perform useful functions.

- `listen:` this creates the `serverSocket` that will listen on the port that was specified in the constructor or by using `setPort`. It also starts this instance as a thread that will, in the `run` method, repeatedly wait for new clients to connect.

- `stopListening:` this method signals to the `run` method controlling the thread to stop looping, and therefore terminate. No new clients will be accepted until the `listen` method is called again. Any connected clients can still communicate with the server because their connections are controlled by separate threads.
- `close:` this does the same thing as `stopListening`, but goes further: it disconnects all connected clients and closes the server socket.
- `sendToAllClients:` this attempts to send a message to all clients.

The methods `listen` and `close` can throw an `IOException`.

Utility accessing methods. These inquire about the state of the server or make modifications to that state.

- `isListening:` determines if the server is listening for new clients.
- `getNumberOfClients:` returns a count of the number of currently connected clients.
- `getClientConnections:` returns an array of instances of `ConnectionToClient` (the array is declared as an array of `Thread`, but `ConnectionToClient` is a subclass of `Thread`, so that you can *cast* the elements of the array to `ConnectionToClient`). You can use this method to write services that do something with all clients, such as searching for clients that have a particular property. This is one of the most important service methods available to the developer of concrete subclasses.
- `getPort:` finds out what port the server is listening on.
- `setPort:` instructs the server to listen on the specified port *next time* `listen` is called; it does not change the port on which the server is currently listening.
- `setBacklog:` sets the size of the queue length. If a client attempts to connect when this queue is full, the connection is refused. The queue can get full if large numbers of clients try to connect, and the server cannot accept them fast enough.

The callback methods of `AbstractServer`

These five methods are all called when important events occur.

Methods that may be overridden by subclasses. These may be overridden by subclasses and are called when events occur that may be interesting to concrete subclasses:

- `serverStarted:` called whenever the server starts accepting connections.
- `clientConnected:` called whenever a new client connects; it provides the instance of `ConnectionToClient` (described below) as an argument.

- `clientDisconnected:` called whenever the server disconnects a client using a call to the close method of `ConnectionToClient`. It provides the instance of `ConnectionToClient` as an argument.
- `clientException:` called whenever a client disconnects itself, or is disconnected as a result of a network failure.
- `serverStopped:` called whenever the server stops accepting connections as a result of a call to `stopListening`.
- `listeningException:` called whenever the server stops accepting connections due to some failure.
- `serverClosed:` called when the server closes down.

In the same way that the client had only one abstract method, the server has only one abstract method called `handleMessageFromClient`. This single slot method is the most important piece of code that a developer of a concrete subclass will write. When called by the framework, it provides as arguments the message received as well as the instance of `ConnectionToClient` corresponding to the client that sent the message.

The public interface of `ConnectionToClient`

For the period of time during which each client is connected, an instance of `ConnectionToClient` exists for that client. The currently existing instances of this class can be accessed using `getClientConnections`, as described above, as well as several of `AbstractServer`'s callback methods. You use such objects to find out information about clients and to communicate with clients.

`ConnectionToClient` is a concrete class. Users of the framework can simply use its facilities – they do not have to subclass it. It provides five service methods that can be used by developers of concrete subclasses of `AbstractServer`. The first two of these can throw an `IOException`.

- `sendToClient:` the central method that is used to communicate with the client.
- `close:` causes the client to be disconnected.
- `getInetAddress:` obtains the Internet address of the client connection.
- `setInfo:` allows arbitrary information to be saved about this client. For example, the concrete server could give certain clients special privileges, which would be recorded using this method. More simply, this method could be used to record the client's user id.
- `getInfo`: allows the retrieval of any information that had been saved using `setInfo`.

How an application developer should use `AbstractServer` and `ConnectionToClient`

A developer who wants to create a server using OCSF needs to perform the following activities, which are almost identical to what a developer of a client needs to do:

- Create a subclass of `AbstractServer`.
- In this subclass, implement the slot method `handleMessageFromClient` to do something useful with any messages coming from the client.
- Arrange for some code somewhere to create an instance of the new subclass of `AbstractServer` and to call the `listen` method.

In almost all servers, the developer will also want to do the following:

- Arrange for code somewhere to send messages to clients, using the `getClientConnections` and `sendToClient` service methods. For a simple server, it might be possible to use `sendToAllClients` instead.
- Implement one or more of the other callback methods to respond in intelligent ways to various events.

A few details of the private internals of `AbstractServer` and `ConnectionToClient`

You can design a server knowing only the above information; however, the following are a few of the internal details of the server side of OCSF. These details will help you form a better understanding of how it works.

- The `setInfo` and `getInfo` methods make use of a Java class called `HashMap`. A `HashMap` can store an arbitrary object using some other arbitrary object as a key. The key can then be later used to retrieve the stored object.
- Many of the methods in the server side of OCSF are *synchronized*. Synchronizing a method ensures that no other thread can access the object while it is running. Since there are many `ConnectionToClient` threads that could all make concurrent changes to the data maintained by the server, synchronization guarantees that critical operations are performed one at a time, ensuring the integrity of the data.
- The collection of instances of `ConnectionToClient` maintained by `AbstractServer` is stored using a special Java class called `ThreadGroup`. This class takes care of automatically removing elements when a thread terminates.
- The server must regularly take a temporary pause from listening to see if the `stopListening` method has been called; if not, then it resumes listening immediately. A design alternative would be to have the `stopListening` method force the listening thread to terminate; however, that would leave the `ServerSocket` in an unstable state. The method `setTimeout` can be used to set the interval between server pauses; it defines the maximum time that the server

will take to stop the listening thread. The default value of 500 ms is suitable for most applications.

3.9 An instant messaging application using the OCSF

To illustrate the use of OCSF, we present here a simple client–server instant messaging system. We call this SimpleChat, and its source code can be found on the book's web site. The version presented here is Phase 1 of SimpleChat. Various project exercises found at the end of this and subsequent chapters ask you to add features to SimpleChat.

The server side of SimpleChat is particularly simple. All the server does is echo messages coming from clients to all the connected clients; thus the class is called `EchoServer`. `EchoServer` itself has no user interface; once started its process must be killed or it will run indefinitely.

As Figure 3.8 shows, `EchoServer` is simply a subclass of `AbstractServer`. The `main` method creates a new instance and starts listening for server connections by calling `listen`. To provide feedback, all the callback methods simply print out messages to the user's console. The `main` methods are underlined since they are static.

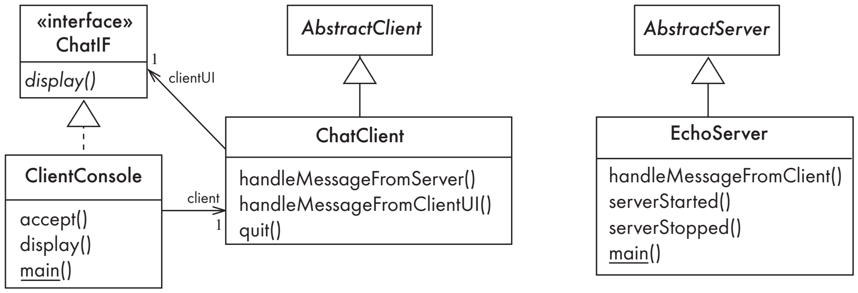

Figure 3.8 **Extending the OCSF framework to build the SimpleChat application**

The `handleMessageFromClient` method does one more thing: it calls `sendToAllClients` in order to echo any messages. The following is the code for `handleMessageFromClient`.

```
public void handleMessageFromClient (
  Object msg, ConnectionToClient client)
{
  System.out.println
    ("Message received: " + msg + " from " + client);
  this.sendToAllClients(msg);
}
```

On the client side, `ChatClient` is a subclass of `AbstractClient` that overrides one method, `handleMessageFromServer`. This method does nothing but arrange for messages to be displayed to the end-user, as discussed below. `ChatClient` also has two other methods that are called by the user interface.

The user interface of the client is carefully separated from the functional part of the client. A Java interface, `ChatIF`, is provided that specifies that any user interface must implement a single method, `display`. One class called `ClientConsole` implements this interface; some other class could be substituted in place of `ClientConsole`. For example, on the book's web site you will find a class called `ClientGUI`. This substitute class simply has to implement the `display` operation to work properly with `ChatClient`.

When the client starts, the `main` method in `ClientConsole` runs. This creates instances of `ClientConsole` and `ChatClient` (which runs as a second thread), and then calls a method called `accept` to await user input. The `accept` method runs in a loop until the program is terminated; it sends all input to the instance of `ChatClient` by calling its `handleMessageFromClientUI`. This in turn calls `sendToServer`. The code for `handleMessageFromClientUI` is written as follows:

```
public void handleMessageFromClientUI(String message)
{
  try
  {
    sendToServer(message);
  }
  catch(IOException e)
  {
    clientUI.display
      ("Could not send message to server. Terminating client.");
    quit();
  }
}
```

Communication coming from the server works as follows. The framework triggers a call to `handleMessageFromServer`. This in turn calls the `display` operation of `ChatIF`, which results in a call to the `display` method in the user interface class `ClientConsole`. The code for `handleMessageFromServer` is as follows.

```
public void handleMessageFromServer(Object msg)
{
  clientUI.display(msg.toString());
}
```

In the exercises at the end of the chapter, you will make some changes to Phase 1 of SimpleChat. In subsequent chapters, you will have the chance to make many other improvements and additions to its design. If you follow all the exercises, you will end up being able to transmit drawings in real time with the program.

3.10 Difficulties and risks when considering reusable technology and client–server systems

Software development organizations should design software that is reusable, and should reuse software whenever possible. In both cases, the goal is to reduce the large cost associated with developing the same thing over and over again. One important approach is to actively look for opportunities in any development project to design a framework instead of designing an entire application.

Unfortunately, there are some important risks involved in both reuse and reusability. Software engineers should always consider these issues as part of the risk management process we discussed in Chapter 1.

Risks when reusing technology

- **Poor quality reusable components**. You have to trust that the technology works properly, and that any problems will be fixed. Unfortunately, the designer of the reusable software might not have followed good software engineering practices, and you may discover major problems. The designer may not have the time to fix the technology, or the technology may be so poor that fixing it adds new problems.
  Resolution. Ensure the developers of the reusable technology follow good software engineering practices and are willing to provide active support.

- **Compatibility or availability not maintained**. Later versions of the technology might be changed in ways that are incompatible with how you have used it. Alternatively, the producer of the technology might go out of business or withdraw it from the market. You may therefore be forced to abandon the technology or modify your applications to stay compatible.
  Resolution. Avoid the use of obscure features of technology. Only reuse technology that others are also reusing. Mandate that reuse should be the rule, but allow exceptions in cases where developers can provide a clear justification.

Risks when developing reusable technology

- **Risk from an uncertain investment**. Developing reusable technology takes time away from developing applications and is therefore a calculated risk.
  Resolution. To ensure the investment pays off, carefully plan the development of the reusable technology, in the same manner as if it were a product for a client. Monitor the success or failure of the reusable software so that you can improve your investment decisions in future projects.

- **The so-called 'not invented here syndrome'.** A framework developed by one set of developers might not be used because others fear it might not be supported.
  Resolution. Build confidence in the reusable technology by guaranteeing support,

ensuring it is of high quality and responding to the needs of the users. (The users in this case are the software engineers who adopt the technology.)

- **Competition**. Reusable technology might not end up being used if somebody else develops competing technology that gains wide acceptance. Being beaten by the competition is a risk in any business; however, with reusable software the competitive forces are often not financial in nature. Several groups may develop similar packages and one may be accepted for reuse merely because its developers are better known or 'market' it better.
  Resolution. Ensure the reusable technology is as useful and as high quality as possible. Advertise the presence and advantages of your reusable software.
- **Divergence**. Several development teams using the same framework may want to change it in different ways.
  Resolution. Ensure that the framework is well tested and reviewed; if it is designed to be general enough, then it will be less likely to suffer from divergent changes.

Risks inherent in client–server or other distributed systems

- **Security**. Distributed systems are particularly prone to security violations, due to the fact that information is transmitted over a network. Communications can be intercepted, or a denial-of-service attack can be implemented.
  Resolution. Recognize that security is a big problem with no perfect solutions. Incorporate encryption, firewalls and similar protective measures into your designs.
- **Need for adaptive maintenance**. If clients and servers are developed by different organizations, then the developers of clients are frequently forced to upgrade their clients whenever the server is changed.
  Resolution. Ensure that all software is forward-compatible and backward-compatible with other versions of clients and servers. Achieving this requires designing the client–server protocols to be very general and flexible.

3.11 Summary

In this chapter we have studied reusable technology, which should be the basis for most software development projects. When developing software, you can reuse many kinds of things, ranging from the expertise of people who have worked on past projects up to complete applications. You should also strive to make anything you develop as reusable as possible.

An important type of reuse is reuse of frameworks. Frameworks are software systems that are not immediately usable, but can be quickly extended to build an application or part of an application, by providing essential details that are missing.

We studied in depth a client–server framework written in Java. The Object Client–Server Framework (OCSF) provides all the essential features of any

Network ethics

People who design and work with distributed systems must develop a heightened awareness of certain ethical issues.

With distributed systems, it is particularly easy to violate people's privacy. This can be done by simply gathering data about people as they use network-based programs, or else by actively intercepting communications. Both these activities should normally be considered unethical unless people have consented to the release of their private information, are able to withdraw that consent easily at any time, are able to examine and correct the information collected about them, and are aware of the method by which the information is collected.

Knowledge of how to develop distributed systems also brings with it knowledge of how to develop harmful programs such as viruses or Trojan horses, as well as how to 'hack' into systems. Some people take a perverse pride in using such knowledge; however, doing so is illegal and extremely unethical, no matter whether the knowledge is used for 'fun' or maliciously.

client–server system. On the server side it includes facilities for starting and stopping the server, maintaining a list of clients, sending messages to clients and responding to messages received from clients. On the client side, it provides facilities for connecting and disconnecting from a server, sending messages to the server, and responding to messages coming from the server.

We showed how it is possible to take this framework and implement only a few methods in order to create an instant messaging system we call SimpleChat.

3.12 For more information

Reuse

- ReNewsWWW: http://frakes.cs.vt.edu/renews.html The Electronic Software Reuse and Re-engineering Newsletter on the World Wide Web
- I. Jacobson, M. Griss, P. Jonsson, *Software Reuse: Architecture Process and Organization for Business Success*, Addison-Wesley, 1997
- C. McClure, *Software Reuse Techniques: Adding Reuse to the System Development Process*, Prentice-Hall, 1997

Frameworks and product lines

- M. E. Fayad, D. C. Schmidt and R. Johnson, *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, Wiley, 1999
- G. Rogers, *Framework-Based Software Development in C++*, Prentice Hall, 1997
- D. F. D'Souza, A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*, Addison-Wesley, 1999

- The product line practice initiative: http://www.sei.cmu.edu/plp/

The Internet, networking etc.

- The Living Internet: http://livinginternet.com. This web site gives an excellent overview about the Internet, including a discussion of IP addresses etc.
- M. Hughes, M. Shoffner and D. Hamner, *Java Network Programming: A Complete Guide to Networking, Streams, and Distributed Computing*, 2nd edition, Manning Publications, 1999. http://nitric.com/jnp/

The client–server architecture

- The Webopedia entry for this topic: http://webopedia.internet.com/TERM/c/client_server_architecture.html
- The client–server newsgroup news:comp.client-server. http://groups.google.com/groups?&group=comp.client-server

Project exercises

The following series of exercises should ideally be followed in sequence. After completion of these exercises you will have built Phase 2 of SimpleChat. A complete implementation of Phase 2 is available on the book's web site.

E48 On the book's web site, you will find a set of 'test cases' for Phase 1 of the `SimpleChat` program. We will discuss test cases in much more detail in Chapter 10. For now, you can simply see them as a set of instructions that allow you to verify the functionality of the system. You can also use them to learn about the system. Pick ten Phase 1 test cases and execute them.

E49 This exercise will help you to become familiar with the internals of OCSF and Phase 1 of an instant messaging application we call SimpleChat. Modify the application to provide the following features (Remember: do not modify the OCSF framework):

Client side:

(a) *Currently, if the server shuts down while a client is connected, the client does not respond, and continues to wait for messages*. Modify the client so that it responds to the shutdown of the server by printing a message saying the server has shut down, and quitting. Design hint: look at the methods called `connectionClosed` and `connectionException`.

(b) *The client currently always uses a default port*. Modify the client so that it obtains the port number from the command line. Design hint: look at the way it obtains the host name from the command line.

Test that this works by connecting a client to a server using a different port from the default. If the port is omitted from the command line, then the default value should still be used.

Server side:

(c) *Currently the server ignores situations where clients connect or disconnect*. Modify the server so that it prints out a nice message whenever a client connects or disconnects. Hint: you will simply have to write code in EchoServer that overrides certain methods found in AbstractServer – study the AbstractServer description above to determine which methods you have to override.

E50 Make further modifications to the SimpleChat application, as follows:

Client side:

(a) *Currently, the client simply sends to the server everything the end-user types. When the server receives these messages, it simply echoes them to all clients*. Add a mechanism so that the user of the client can type commands that perform special functions. Each command should start with the ‘`#`’ symbol – in fact, anything that starts with that symbol should be considered a command.

You should implement commands specified as follows:

(i) `#quit` causes the client to terminate gracefully. Make sure the connection to the server is terminated before exiting the program.

(ii) `#logoff` causes the client to disconnect from the server, but not quit.

(iii) `#sethost <host>` calls the `setHost` method in the client. Only allowed if the client is logged off; displays an error message otherwise.

(iv) `#setport <port>` calls the `setPort` method in the client, with the same constraints as `#sethost`.

(v) `#login` causes the client to connect to the server. Only allowed if the client is not already connected; displays an error message otherwise.

(vi) `#gethost` displays the current host name.

(vii) `#getport` displays the current port number.

Server side:

(b) *Currently, the server does not allow any user input*. Study the way user input is obtained from the client, using the `ClientConsole` class, which implements the `ChatIF` interface. Create an analogous mechanism on the server side. Design hint: you will have to add a new class you can call ServerConsole that also implements the ChatIF interface. Following your modifications, the following should be true:

(i) Anything typed on the server's console by an end-user of the server should be echoed to the server's console and to all the clients.

(ii) Any message originating from the end-user of the server should be prefixed by the string 'SERVER MSG>'.

(c) In a similar manner to the way you implemented commands on the client side, add a mechanism so that the user of the server can type commands that perform special functions. You should implement commands specified as follows:

(i) #quit causes the server to quit gracefully.

(ii) #stop causes the server to stop listening for new clients.

(iii) #close causes the server not only to stop listening for new clients, but also to disconnect all existing clients.

(iv) #setport <port> calls the setPort method in the server. Only allowed if the server is closed.

(v) #start causes the server to start listening for new clients. Only valid if the server is stopped.

(vi) #getport displays the current port number.

E51 Make further modifications to the SimpleChat application, as follows.

In Phase 1, clients are always anonymous. When a message is sent from a client, it is echoed to all the other clients, but nobody knows who sent it. In this exercise, you will implement a basic mechanism by which clients have a 'login id' that is known both to the client and the server.

Client side:

(a) Add a new 'login id' command line argument to the client. This should be the first argument, before the host name and port, because the host name and port are optional in the sense that if they are omitted, defaults are used. The login id should be mandatory; the client should immediately quit if it is not provided. Design hint: the login id should be stored in an instance variable in ChatClient. You might ask the question: why not put the instance variable in ClientConsole? The reason is to separate the user interface (how information is displayed and input) from the other aspects of the system.

(b) Whenever a client connects to a server, it should automatically send the message '#login <loginid>' (i.e. the string #login with the login id appended to it) to the server. Note that this use of the '#' is different from what we have seen so far: the `#login` is sent to the server; it is not handled by the client as was the case with `#quit`, `#logoff` etc.

Server side:

(c) Arrange for the server to receive the #login <loginid> command from the client. It should behave according to the following rules:

(i) The `#login` command should be recognized by the server. Design hint: modify `handleMessageFromClient` so that it does more than just echo messages.

(ii) The login id should be saved, so that the server can always identify the client. Design hint: use the `setInfo` method to set the login id and the `getInfo` method to retrieve it again later.

(iii) Each message echoed by the server should be prefixed by the login id of the client that sent the message.

(iv) The `#login` command should only be allowed as the first command received after a client connects. If `#login` is received at any other time, the server should send an error message back to the client.

(v) If the `#login` command is not received as the first command, then the server should send an error message back to the client and terminate the client's connection. Hint: use the method called `close` found in `ConnnectionToClient`.

E52 Now that you have completed Phase 2 of SimpleChat, you can execute the test cases provided in the web site for Phase 2. You should execute all the test cases that are indicated to apply to Phase 2, along with a sample of test cases that are marked as relevant only to Phase 1. When testing, use your own server with somebody else's client and vice versa. If you have followed the instructions above consistently, then you should have no trouble doing this.

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

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

6.14 Enhancing OCSF to employ additional design patterns

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

Client connection factory

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

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

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

To enable the server class to instantiate an application-specific subclass of `ConnectionToClient`, OCSF provides an optional Factory mechanism. There are two keys to this. The first key is an interface called `AbstractConnectionFactory` (see Figure 6.15). You create an application-specific factory class that implements the createConnection method in this interface. Your factory class will in turn create instances of your own subclass of ConnectionToClient. The second key is the method setConnectionFactory found in AbstractServer. Your server class calls this to ensure that whenever a new client attempts to connect, your factory will be directed to instantiate your subclass of ConnectionToClient to handle the connection.

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

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

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

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

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

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

Observable layer

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

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

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

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

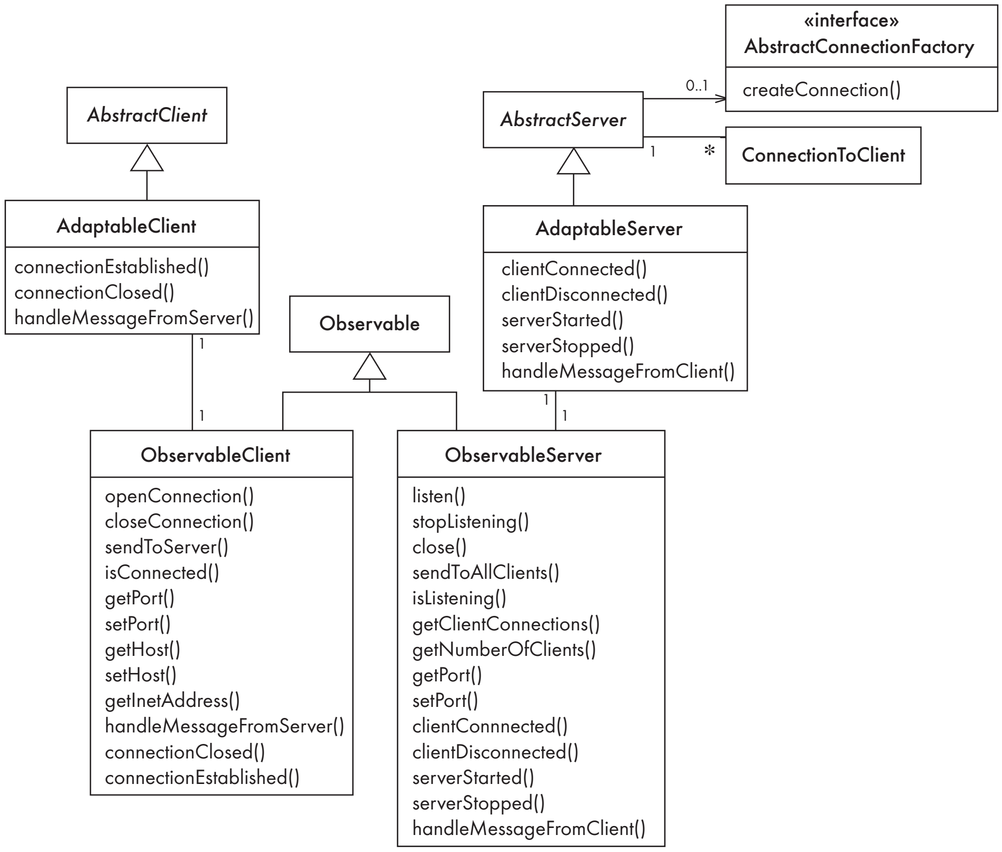

Figure 6.15 **The Object Client–Server Framework with extensions to employ the Observable and Factory design patterns**

Implementation of the Observable layer

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise

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