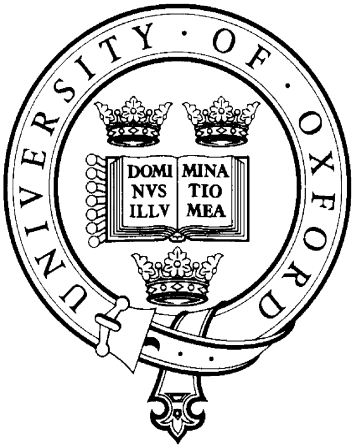
Advanced Diploma in Data and Systems Analysis

Software Construction using Objects

(Object-oriented analysis and design notations & architectures)

Amended, fifth edition.



http://datasystemsanalysis.conted.ox.ac.uk//

Robert Lockhart

Department for Continuing Education
Oxford University

Table of Contents

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

Tony Hoare – one of the founders of theoretical computer science and fellow of Kellogg College Oxford, 2009.

Chapter 1: The Software Development Process

Software development has such a popular reputation for inefficiency and failure that it might be worth pointing out that no one would embark upon it were it not for the huge success of so many computer systems.

Academics and practitioners have done a great deal of work, of very variable quality, on the *process* of constructing software. Development methodologies and philosophies have swung in and out of fashion over the past 30 years, attended by heated debates and controversies.

In this course, we attempt to give you advice and information that is coloured by a common-sense, and practical, view of the subject. We try to explain various ideas and techniques as clearly as we can without subscribing to a specific discipline. This might cheat you of some of the coherence of particular viewpoints but at least permits you to grasp the key ideas and make up your own mind.

You may subsequently find yourself in an industrial or educational setting where a particular methodology is pursued more rigorously and you may then reappraise some of what we tell you here. So, don't imagine that anything we say is the final word on a subject or that our account describes the only way in which these ideas can be applied. It is entirely possible to encounter situations in which people advocate particular methodologies or viewpoints with an almost messianic zeal. Without commenting on the validity of such attitudes, it might be said that one of the very few benefits of age and experience is our knowledge of occasions in the past in which this has occurred, together with subsequently readjustments often bordering on abandonment. Take the recurrent "magical bullets" of software engineering with a pinch of salt!

There are many occasions when we mention an important topic and then say that we have no space to treat if properly. We hope this is not too irritating to you, we obviously have a limit on how much we can do in this short course and we want to bring to your attention other important issues that you could perhaps research on your own. At the very least, note these topics and bear them in mind after our course has finished. Part of the idea of this course is that it never ends, it merely starts you on a journey.

Processes

Software construction entails a development **process**. That is, it involves activities that take place over time - activities which result, we hope, in some sort of finished **product**.

The distinction between process and product is very useful.

In particular, a development *process* has a life of its own. One needs to manage it; and management involves the collection and interpretation of measurements. Many of the measurements that one can collect are associated with the artefacts, the products, produced by the development process.

These products may be final deliverables of the project – code modules, databases, and so on; or they may be intermediate artefacts produced as part of the construction process - diagrams, textual statements, data definitions, pseudocode, staff attendance data, or whatever.

The associated measurements may be fairly simple and involve little more than the time taken to produce the artefact or the number of subsequent revisions or bugs. Alternatively, they may be more complicated and relate to how convoluted diagrams are, or how closely the major structures of the system relate to one another.

Measurements of the sort we are discussing are very often called **software metrics**. In this simplified account of the subject you can take it that most of our metrics relate to artefact production and are therefore **product metrics** but other sorts of metric, such as **process metrics**, are important. You will find a good deal of web information on them and there are many books on software measurements, some of them very nearly readable!

Managerial decisions made without recourse to scientific measurements are always likely to be wrong. Software managers need some knowledge of the measurement resources available to them. They need to know what can be measured, and they need to know what cannot.

Processes are complicated and organic entities and during their life unforeseen problems will arise and difficult decisions need to be made. Politics and litigation can cloud issues in destructive ways. In this account we have space to tell you a little of what needs to be or can be done, but not about measuring or managing it.

An illustration

Some years ago, one of the developers of this course worked in a large government research institution which decided to commission a new financial system from an internationally famous software vendor. The story in short is that the system, at least initially, was a complete and embarrassing failure.

The reason for the failure is more complex than you might imagine. It was certainly possible to isolate a whole raft of things that were, in some sense, done wrongly. For example, the new system replaced the old one in a single day. The old one was irretrievably switched 'off' and the new one 'on' (whereupon it fell over!). Any software development guru would laugh at such a scenario.

Except that the constraints of the problem dictated that this policy of junking the old system was essential. A rigid software methodology that forbad that would simply be howling at the moon!

The impressive aspect of the failure was that the research institution did so much that was in some sense 'correct'. As one example of this, they instituted a structured way to classify problems and deal with them, and, day after day, software engineers addressed these problems, and solved them.

The trouble was that these day-to-day successes gave a wholly inappropriate feeling that the project was reviving. In fact, the daily *rate* of new problems[1] was undiminished in both the short and long term, and design flaws meant that the act of solving one problem very often resulted in the production of a new one.

[1] This is one example of a *process* metric. Other examples include such things as the mean time to failure of the system and the number of people working on it.

If this story has a moral it is that any complicated project needs someone who has an overview of what is going on, and some resources of old-fashioned common sense![2]

We might add that contract awards based on simple lowest cost tendering are very often not in the public interest and equally often, remarkably corrupt.[3]

A functional view

We consider the development process in terms of what needs to be done, rather than how that is to be managed. Our emphasis is related to the production of various products associated with the development process.

And the development processes that concern us here are associated with the production of **object-oriented systems**. We expect that you have had some previous exposure to objects but present a brief, idiosyncratic, overview of them in our next chapter.

Processes have a structure of their own, relating to the sequence of activities, how activities relate, and what needs to be done. They are systems in their own right and have their own life cycles.

People have consciously attempted to manage processes by relating them to specific **life cycle models**, which often have the following interrelated aspects:

- requirements capture;
- high-level analysis and specification;
- modelling and design;
- prototyping;
- implementation;
- testing;
- delivery;
- maintenance.

You will already know that these concerns are rarely treated sequentially and that project development is more organic and iterative than might be suggested by this flat list. Nevertheless, it can be useful to imagine distinct areas of interest, in both organisational and methodological terms.

You also know how development methodologies relate to life cycle models by isolating areas of interest and relating them to product deliverables associated with each of these stages.

Areas of interest associated with methodologies include:

- data flows and processing (deliverables might include data flow diagrams and pseudocode);
- data structures (deliverables might include entity-relationship diagrams);
- system behaviour (deliverables might include functional specifications and entity life histories);
- interfacing (deliverables might include storyboards, walkthroughs and decision tables).

[2] The message is: 'Don't become so obsessed with technique and methodology that you fail to see the blindingly obvious about what you are doing! Don't try to dig your way *out* of a hole!'

[3] Tendering in which a number of very different organisations proffer bids which are virtually identical is often a good example of this. The interested reader might investigate telephone franchise bidding, and rail contract bidding, in the United Kingdom.

One's choice of techniques is based on the nature of the problem to hand, the experience of one's staff, and the facilities available. In practice, all these ideas are important and a methodology that exalted one at the expense of another would be quite useless.

The products that primarily concern us encapsulate aspects of the *analysis* that we perform on the problem and our response to that analysis in design terms. Our aim is to give you some skill and understanding of these products because they form the scaffolding of the software edifice that is being constructed.

You should note immediately that these products are not usually the key deliverables of a software engineering product. They are "way products" constructed in the process of reaching the desired deliverables. We hope eventually to produce a sister course that continues the story into the realms of implementation[4].

Technical communication

The development of computer systems is a skilled and creative activity, often involving a considerable number of people. Development teams need to 'speak the same language'. They must be able to formulate a common approach to the analysis of the system, and communicate technical ideas and design decisions.

This sort of communication ranges from the shorthand notes you might write to yourself to note a brainwave; through the brief, technical descriptions you would use to agree analysis and design decisions for your colleagues; right on to the more formal documents that represent the technical understanding of what is wanted, which are exchanged between your software company and the organisation commissioning, and paying for, the project.

Historically, this sort of communication first used the shorthands associated with formal programming - pseudocode being an example of that.

From the 1970s, more complicated, purpose-built notations began to emerge - such things as Jackson structured diagrams and formal specification languages. We might loosely describe these ideas as **modelling techniques**. They describe key aspects of the software problem that we are attempting to solve and can be used to inform and control the process of implementing our designs.

There is a raft of such techniques and notations. A good software developer will need to have some familiarity with many of them but will probably use only a small subset of these. Some development contracts even insist on the use of particular models or techniques.[5]

We expect that many people reading this will have previous experience of other modelling techniques, such as those discussed in our previous units. We shall not assume any such prior experience here but we shall, occasionally, relate a particular technique we are describing to ones that fall outside our present scope. Readers with no familiarity of the techniques in question can safely ignore such asides.[6]

[4] This has been our honest desire for some years, but the prospect seems increasingly remote.
[5] As one instance of this, in the heyday of formal specification languages the US Department of Defence insisted that all new software developments should be formally specified. You might like to know that this requirement has now been dropped!
[6] So, if you have really never come across entity-relationship diagrams or data flow diagrams, you really don't have to worry! Although we might worry about what you were doing in the first three units of this course!

Models are not systems

One hopes that the delivered system is as is specified by our models but this needs to be tested in the course of constructing them. The testing process associated with that is sometimes called **validation.** Validation asks - “are we building the right system?”. That is, one agreeing with the specification.

A related testing process asks whether the correct system has been successfully implemented - “did we build the system properly?”. That is the process of **verification**.

The situation is similar to that which a professional programmer encounters when reading the annotation of complicated code. You may understand the annotation and even know what the code is supposed to do, but the only arbiter of whether it does it is the code[7] itself. What someone tells you they have done is very often distinct from what they have actually done. All such claims need *verifying*.

Validation treats concepts; verification treats realisations; that is: validation involves pre-implementational testing, verification involves post-implementational testing!

That is not to say that validation has to occur before verification because in practice both consideration do arise throughout the development process; but it does mean that, in this unit validation is of greater concern to us.

Software modelling

Software models are abstractions used in the representation or development of software. They are a little different from the models used in the physical sciences because they:

- focus on one single aspect of a problem, or perhaps a limited view of it, at the expense of other aspects or views;
- *represent* rather than *explain*;
- tend to be constructed as grammatical assemblages of recurrent simple primitive notions.

Consider pseudocode as an illustration of these points.

- Pseudocode describes processing, not interfacing or data storage.
- Pseudocode illustrates what is to happen at the local level rather than saying anything about the overall system.
- Pseudocode usually conforms to clear, if unstated, grammatical principles.[8]

Syntax

The primary point of a software model is to communicate technical ideas to humans. Most software models do not need to have the syntactical purity of a program written in a programming language because most software models do not have to be processed by machines.[9]

Nevertheless, modelling notations do have a grammar, and a model will not really be intelligible unless your notation adheres to it[10].

[7] The commonly expressed view that annotation tells you what the code does is laughable in practical terms. One writes annotation with that in mind, but so often one deceives oneself.
[8] If it did not, it would not serve its principal purpose – which is communication.
[9] There are exceptions to this but they do not concern us here.
[10] A readiness to depart from precision in notations is the sign of a bad engineer!

Most of our modelling notations use the graphical representations associated with the **Unified Modelling Language (UML)**. This is an extensive set of graphical notations, widely used in the description of software – particularly, object-oriented software[11].

Use case modelling

When considering a system one might seek to isolate the various categories of users who would interact with it in terms of their capabilities (system managers having more privileges than ordinary users, and so on) and the possible things they might wish to do, including the possible system responses (login, fail password, logout, etc.)

The models we would construct to capture these ideas are called **use cases**. Such models relate to *what* we wish to happen, but say nothing about *how* it is to be achieved. They tell an interesting part of the story without attempting to cover all of it. One might say that the use case stories relate to *who* does *what* and *for what purpose*. Except that the "who of it" (the "Actors") may not necessarily be human!

Use cases are important to us and we shall have a lot to say about them. They are so deceptively simple-minded that the beginner is in some danger of missing the point. Any software system is likely to involve quite a complicated battery of possible use cases. The important thing about modelling is not to record them all in tedious detail, but to capture the main ones as simply as possible.

A software model should represent something, simplify it, and communicate it. Models abstract important aspects of a problem and record the decisions reached about those aspects, but they almost always deliberately conceal other aspects of the situation from immediate consideration.

Complicated and unintelligible software models are worse than useless. Almost the entire point of software modelling is communication. So if your model is so complicated as to be impossible to understand, it is stupid. Sadly, there are historical examples of institutions positively encouraging such stupidity.

> When you use UML, make sure that your diagrams make sense as UML diagrams – that is, that they are syntactically correct and use the correct symbols, in the correct way.

Although we explain many aspects of UML diagrams, we don't feel that it is our job to give you a precise definition of the whole of the language. Such definitions are freely available on the web and there are many good books that describe UML too. You should certainly consult other sources about UML. Our emphasis is on using it practically within a particular setting.

[11] If you are already fed up with the company line on UML, you might like to do a web search for an article called "13 reasons for UML's descent into darkness."!

Process and system diaries

The construction of this unit is a process involving a principal author, a second author and at least five other people. The principal author maintains an informal 'process diary' associated with the undertaking and recording notes, difficulties, comments, ideas and references.

This diary is an intermediate product of the development process. If all goes well, you should be largely unaware of its existence. The diary exists as handwritten notes; comments attached to the principle *Word* documents using the *Word* comment facility; an e-mail folder containing advice and responses from members of the Course Team; and annotated course material from other courses given by the Course Team and the other academic institutions.

Although we have said that we are more concerned with product than process here, we do strongly recommend that you keep your own 'diaries' in any process that concerns you. This applies both to the design examples you will encounter here, and the main process that shall occupy you now – studying the unit itself![12]

> Write notes, even to yourself, with attention. You are addressing people quite distinct from yourself: either other individuals, who need to read your handwriting and understand your syntax; or an older and different you, more experienced, and perhaps with no clear memory of the note you are making now, but burdened with new and different understandings of the terms you are likely to be using.

In what follows we shall be describing the development of an object-oriented system and relating the techniques that we introduce to a particular case study. We shall occasionally refer to a **system diary** which, we shall assume, will contain informal notes that are to be heeded as the system develops.

The form and structure of system diaries of this sort is something that we shall leave to your own good sense. Our advice is as follows:

- don't make them too complicated - no-one will use them;
- make them intelligible and available to everyone on your team;
- don't be slapdash or colloquial about what you record;
- keep them as a diary, dating each entry and mentioning something about its context.

[12] However hard we may try to explain something, it will probably be more intelligible to you if expressed in your own language. Part of the work of studying a course like this is a translation process from the way we express things to the way you do.

© OUDCE, January, 2013

Chapter 2: Objects

Some history

We start this chapter with a brief, historical account of how the ideas associated with object-oriented programming first arose. We go on to a rather brisk summary of object-oriented ideas.[13]

It seems appropriate to illustrate some of these ideas using snippets of Java code and we try to couch it in such a way as not to assume prior knowledge of the Java language.

We choose Java for a number of reasons. It is one of the most popular[14] object-oriented languages; its syntax is very representative of other popular languages; and its treatment of objects is more direct and less complicated than, for example, that of C++[15].

We could as easily have chosen Smalltalk or C++ for our illustrations because the tale we tell is expressed in very similar ways in these languages.

The first programs were sequential arrangements of electronic switches. High-level programming languages arose well into the history of computing and can be thought of as another step in a journey of abstraction and representation that continues to this day.

As the business of programming became more complicated, and computers more sophisticated, abstraction and representation were extended to the data on which programs operate. File and data structures, relational databases and, finally, objects entered into the central concerns of computer science.

These data abstractions can hardly be viewed solely in terms of arrangements of binary data. They are actually combinations of the data and its means of access. Indeed, it has become fashionable to view data structures almost exclusively in terms of the way in which they are *processed* – paying little attention to *how* data is actually arranged in store or in memory.

One might regard relational databases as the ultimate example of this viewpoint. Most people need very little specific knowledge of the internal structure of their database systems and the same means of access, encoded in SQL statements, applies to the relational databases offered by all leading software companies.

In short, the insight usually associated with von Neumann, that code instructions should be treated as data, applies the other way round. Data is inseparable from the operations one may perform on it.

The evolution of our present ideas about objects has a number of sources other than this philosophical view of information storage. Precursors include various attempts at increasing the accessibility of programming and specific needs relating to computer simulations. The story is interesting but not one we have time for here. As an aside, we observe that views of objects are still evolving.

---

[13] This account is not intended for readers with absolutely no previous experience of object-oriented ideas. Those who are totally innocent of object-oriented systems may find that some independent background reading would be helpful. There are many good tutorials available on the web and any introductory book on Java is likely to cover this topic.

[14] Popularity, in computing, is not always due to utility. One clear reason for the emergence of Java is that it is essentially free, although of course the tools and development environments associated with going over to Java can be quite expensive!

[15] Our first version of this unit attempted to be much more independent of Java and some feature of what we say are a little distinct from the Java viewpoint.

© OUDCE, January, 2013

One aspect of modern object-orientation that does impinge on our present concerns is the fact that early work on objects was closely associated with the invention and development of what we would now describe as **graphical user interfaces**.

This is an association that continues, with respect both to the tool-kits available to software developers and to the human interfaces associated with delivered systems. It colours the analysis and design of object-oriented systems to this day.

Object-orientation

Object-orientation views computer systems as comprising cooperating assemblages of data called **objects**. These additionally possess internal processing capabilities that define the data manipulations appropriate to the information they contain.

The processing capability associated with an individual object is usually called that object's **behaviour**.

Object-oriented processing is entirely considered as involving individual objects that pass communicating '**messages'**[16] to one another. Each message maps to a particular, predefined, piece of behaviour, a '**method'**.

We speak of the internal data of an object as that object's **attributes**. So, our rather loose[17] characterisation of objects is as software structures possessing *attributes* and *behaviour*.

These terms give a strong clue as to why object orientation is felt to be so appropriate in the production of systems modelling aspects of the real world such as:

- hospital administration systems (where objects might be wards, patients, prescriptions, duty rosters, doctors, beds, drugs, etc.);
- biological populations (where objects might be individual animals, predators, environments, etc.);
- shopping malls and purchasing (where objects might be individual firms, product catalogues, shopping carts, orders, customers, etc.).

We shall refer back to some of these examples later in this unit.

Each object has a set of methods that can be appealed to by other objects. This is achieved via the message-passing that we referred to earlier. Indeed, the only way[18] to get the data held by the object to be processed is to send that object a message which invokes one of the very special methods which are directly associated with message passing.

These special methods have a distinguished role in communication with the object and are said to comprise that object's **interface**. Observe that an interface defines all the direct dealings that one can have with a given object.

[16] In our experience, beginners can become confused and disheartened by this term 'messages'. In practical terms it is no more sophisticated or confusing than a standard procedural call in imperative programming – and it is usually hard to distinguish from that at first.

[17] Loose, in so far as it is informal. The critical reader will observe an embarrassing circularity in most accounts of objects. We believe that our account is appropriate for our present needs and that further precision would not be helpful at this time.

[18] At this point the world-weary reader might realise that some object-oriented languages permit one to access an object's data in other ways. This is generally regarded as bad form in object-oriented circles and we shall ignore it here. See "not respecting encapsulation", below.

© OUDCE, January, 2013

Of course, a message to an object might result in the object appealing to its own internal methods for service or processing reasons. These internal methods might not be directly accessible to other objects,[19] and so might not be considered to be part of the object's interface.

A large part of what constitutes an object is invisible to other objects or the system at large. Dealings with that object are precisely on the terms laid down by its interface. This is the principle of **encapsulation**[20]. It permits programmers to develop objects in relative isolation, with their internal workings only apparent to the people actually working on the object in question and everyone else viewing it as a "black-box" of prescribed services.

Encapsulation localises and identifies responsibility and encourages a modular approach to software development in which objects provide services without the user needing to worry about how they do it.

Three uses of the term 'interface'

This unit will use the term 'interface' in at least three distinct, but related, ways.

2.1 As such methods belonging to an object as may be invoked directly as messages from other objects (the sense use above).
2.2 As the way in which a particular delivered system interacts with its users - most commonly a series of linked visual frames - but all the other possible ways too.
2.3 As a structural concept of the Java programming language (this comes up in chapter six).

This *overloading* of terminology is unfortunate, but standard.

Summary

1. An object is an assemblage of structured data plus the processing that one can perform on it.
2. One communicates with an object by sending it a message.
3. Objects communicate with each other in precisely the same way.
4. The internal data associated with an object are the object's attributes.
5. The processing associated with an object is encapsulated as discrete chunks of code called methods.
6. Messages work by invoking corresponding methods of the object to which the message is sent.
7. The interface of an object defines the messages that can be sent to it by other objects; and, by extension, the associated methods that are invoked by those messages.
8. The principle of encapsulation dictates that these interface methods are the only way to have dealings with an object.
9. Interface methods are therefore an important subset of the object's methods.
10. Object-oriented systems consist of collections of cooperating objects.
11. Processing works by objects sending messages to one another and responding to the replies.

[19] In Java they might well be declared as **private**.
[20] And the sort of inappropriate and direct access to object attributes that we just mentioned is sometimes referred to as **not respecting encapsulation**.

© OUDCE, January, 2013

Object-oriented languages

Following the work of Turing, Church and others, our present belief is that all programming languages are essentially equivalent[21]. Anything that can be programmed in one language can be programmed in another. This is sometimes called the **Church-Turing thesis**.

This may very well be true in some abstract sense, but it is certainly the case that some languages are more suitable for some things. The object-oriented languages are a group of programming languages that have built-in support for the ideas of object-orientation. These simply have *ready-made* facilities for implementing object-oriented systems but such systems could be constructed in languages that don't have these facilities – it would just be more difficult because you would have to build the facilities first.

"Object-oriented" languages include Java, C++, Smalltalk and the later versions of Visual Basic; whereas Prolog and FORTRAN would generally be regarded as non-object-oriented.

Although each object-oriented language offers direct support for objects, they offer distinct interpretations of what object orientation actually involves, albeit with broad general agreement as to the key features.

State and state-related behaviour

Object behaviour can be complex. The most basic things that one can imagine an object doing are.

- Modifying the present value stored in an attribute.
- Reporting on what the present value stored in an attribute is.

Both of these kinds of behaviour involve accessing attributes. They are sometimes referred to as **set** and **get accessors**, respectively.

The current values contained in an object's attributes (internal data) forms the present **state** of the object.

A set accessor changes the state of the object. A get accessor does not – it simply reports part of the present state of the object.

Object behaviour – that is, the processing defined by the object's methods – usually depends on the object's state. We say that behaviour is **state related**.

Set accessors change state but don't usually need to obtain data. Get accessors obtain data but don't normally alter state.

In practice, most interface methods in modern programming languages would both change state and return a value. The rigid distinction between setting and getting is often idealised in design but not fully realised in practical applications.

More than that, when one sends a message to an object, a method is executed. That method may send further messages to other objects. This means that state changes can occur not only in the recipient object but also, potentially, in every other object of the system: send a message, and the Universe changes!

This unfortunate fact tends to be glossed over in object-oriented programming, and it makes formal reasoning about object programming much more difficult than is the case in, say, procedural or declarative languages. Most object-oriented design methodologies advise limits on

[21] In point of fact, there are indeed "languages", such as SQL, which are actually weaker than the Turing standard language but the principle holds for standard high-level languages.

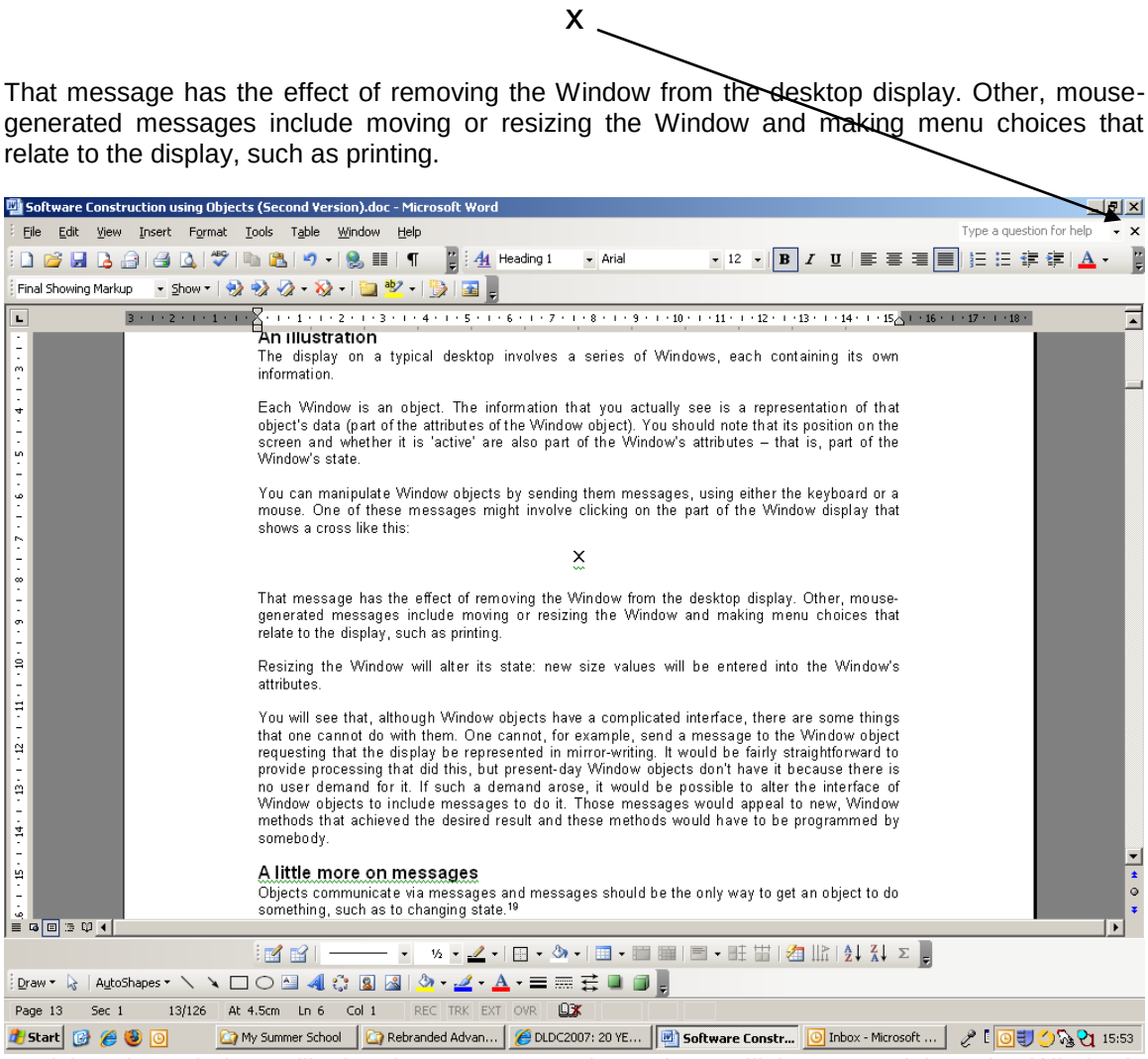
the side-effects associated with message sending and one of the most famous of these is called the **law of Demeter** which is an attempt to limit side-effects of message requests. You might like to find out more about that and similar ideas.

An illustration

The display on a typical desktop involves a series of Windows, each containing its own information.

Each Window is an object. The information that you actually see is a representation of that object's data (part of the attributes of the Window object). You should note that its position on the screen and whether it is 'active' are also part of the Window's attributes – that is, part of the Window's state.

You can manipulate Window objects by sending them messages, using either the keyboard or a mouse. One of these messages might involve clicking on the part of the Window display that shows a cross like this:



That message has the effect of removing the Window from the desktop display. Other, mouse-generated messages include moving or resizing the Window and making menu choices that relate to the display, such as printing.

Resizing the Window will alter its state: new size values will be entered into the Window's attributes.

© OUDCE, January, 2013

You will see that, although Window objects have a complicated interface, there are some things that one cannot do with them.

One cannot, for example, send a message to the Window object requesting that the display be represented in mirror-writing. It would be fairly straightforward to provide processing that did this, but present-day Window objects don't have it because there is no user demand for it. If such a demand arose, it would be possible to alter the interface of Window objects to include messages to do it. Those messages would appeal to new, Window methods that achieved the desired result and these methods would have to be programmed by somebody.

A little more on messages

Objects communicate via messages and messages should be the only way to get an object to do something, such as to changing state.[22]

Messages are appeals to particular methods that form part of the object's behaviour and interface. Messages often enclose information and frequently receive information back.

The following instruction is fairly typical of messages in Java. It involves sending a message to a Java object and storing the reply. This particular message invokes a method called getSize(). The value sent back as a reply to this message is then stored in the variable called frameSize.

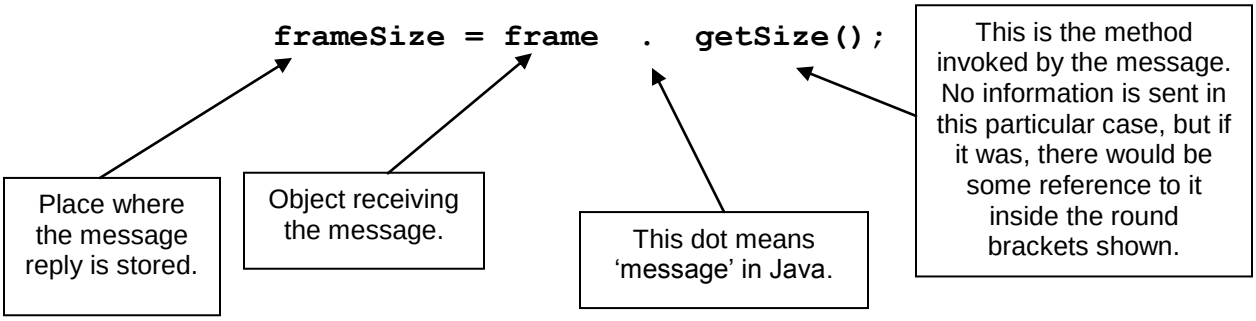


Figure 2.1: A Message in Java

The message itself is coded in the expression: **`frame . getSize();`** .

Like all Java messages, this message causes information to be returned from the object to which it is sent.[23] In this case, that reply is stored in a location referenced by the term **frameSize**[24].

The message is a get accessor. The programmer who wrote its associated method, `getSize`, was careful to name it in such a way as to underline that fact.

In this particular case, **`frameSize`** is one of the attributes of the object that generates this message;[25] and the code snippet itself would probably occur as part of a method of that object.

[22] Readers with prior experience of object- oriented programming will recognise that these assertions are more or less true in individual languages and that language implementations usually involve some compromises with object-oriented ideas.
[23] Java constructor messages, which we shall deal with in a little while, do not, strictly, return data.
[24] This is the part of the code that you read first but which is executed last, the part that involves Java's assignment operation, indicated by the equals sign.
[25] Strictly speaking, a reference to the returned object is stored!

© OUDCE, January, 2013

A number of objects are associated with the expression given in Figure 2.1. These include:

1. The object whose method includes this line of code. We shall call this the **sender**.
2. The object referenced by the variable called frame. This is the object to which the message is sent and we shall call it the **receiver**.
3. The object returned by the message.

The variable frameSize is an attribute of the sender object. After the message completes, it may be used to refer to the object sent back as the message reply.[26]

This message does not itself change the state of the receiver, but the act of storing the reply in frameSize does.

This processing would probably occur within a *method* belonging to the receiver and as such would form part of its internal processing. Outsiders could only change its state by sending it a message.

Of course, messages normally include data. In Java, one might send data in a message like this:

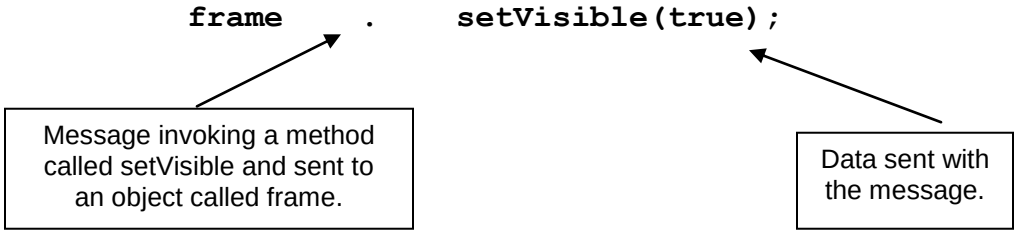


Figure 2.2: A State-changing Message in Java with a Parameter

This particular message causes the frame in question to be visible. (It was previously hidden.) The message sends the boolean data value: **true**.[27]

The data sent with a message is sometimes referred to as that message's **parameters**. You may only send data in this way if the associated method has been coded to expect it (as **setVisible** has). The data sent will often be a reference to one or more objects in the system.

setVisible is a set accessor and the programmer who wrote the associated interface method gave it a name that reflects that.[28]

A little more on methods

Methods are the units of code in object-oriented systems. Each message will map to a particular method that includes the code that is to be run, which may well include further messages to other objects or appeals to other methods associated with the existing object. A typical method consists of a sequence of code instructions and achieves a particular service.

[26] In this particular case, the object sent back is an integer – a Java int. For technical reasons, an int is not regarded as a true object in the Java programming language but we shall ignore this inconvenient fact for now!

[27] The Java programming language has a data type called boolean. This can have one of two values, true or false. In this document, we do no capitalise the first letter of boolean, even though the word is derived from the surname of Charles Boole - i.e. we follow the Java convention.

[28] Like all Java messages, this one does return a value but that is unimportant to the program and so disregarded.

© OUDCE, January, 2013

Methods may have their own internal variables. These are distinct from the attributes that objects have and provide the local storage that most algorithms require.

A lot of the work that object-oriented programmers do is to do with designing and coding methods.

Here is an example of a Java method.

```
public double setPoints()
{ double temp;

  if(x1 == 0) temp = 0;
  else temp = -1;
  return temp;
}
```

Note these points.

- The method is called setPoints().
- It is associated with no parameters (any it had would be declared in the round brackets on the first line).
- It returns values which are of type **double**.
- temp is a local (internal) variable of the method.
- The method tests the current value of x1 and sets temp to 0 or -1 depending on whether x1 is zero or not.
- x1 is an attribute of the object associated with this method. It would be declared somewhere outside this method, as part of the class specification of objects from the class.
- The method exhibits 'state-related' behaviour, that is, it depends on the value of x1.
- The value of temp is sent back as the message reply.
- The method code is contained within the curly brackets. We shall sometimes refer to this code as the method's **specification**.

The position of the declaration of the variable called temp indicates that it is not an attribute of objects from the class. It is a local storage area, something which is entirely internal to the method.[29]

Storage areas and references to them form a complicated series of ideas in programming. In Java all *variables* have an associated **scope**. In this particular case, for example, the variable called temp only has existence within the method. It could not be used, accessed or manipulated from any other part of the program.

Attributes can be recognised because they are defined *within* class declarations but *outside* method definitions. So, temp is clearly not an attribute, because it is defined within the method called setPoints.

You might get setPoints to run by sending a message like this:

```
myObject . setPoints();
```

[29] Some people might describe temp as being an **automatic variable**. In Java it should be thought of as existing only while the method code runs and, as such, it may not be used to hold a permanent value – attempts to do that will come to grief since it only has existence in those moments in which the method code is running.

© OUDCE, January, 2013

and, if you wanted to store the reply sent back (which is the value that temp was given in the method code) you might do it by adding an assignment instruction to this line of code, like this:

```
myFloat  = myObject . setPoints();
```

Java messages must always return values but sometimes a method is not easily associated with a particular return. If there is no important value that has to be sent back, a Java message can be defined so that it replies with a **void** value. This is really just a technical way of saying 'no reply is really wanted although formally we have to declare one'.

It is usually a good idea to keep the code in methods reasonably short. A method should do a single recognisable thing, appealing to other methods for particular services.

There are occasions when this recommendation can be flouted but, as always, you need a good reason for doing that. Long methods are difficult to understand, hard to maintain, and liable to contain programming errors. Some object-oriented design methodologies prescribe short methods as essential to good design.

Method signatures

The first line in the declaration of a Java method specifies these things:

- what the method is called
- what parameters the method expects
- what values the method sends back
- whether it is an interface method or not

This first line is sometimes called the **signature** of the method. It gives all the information needed to use it.

Here is an example.

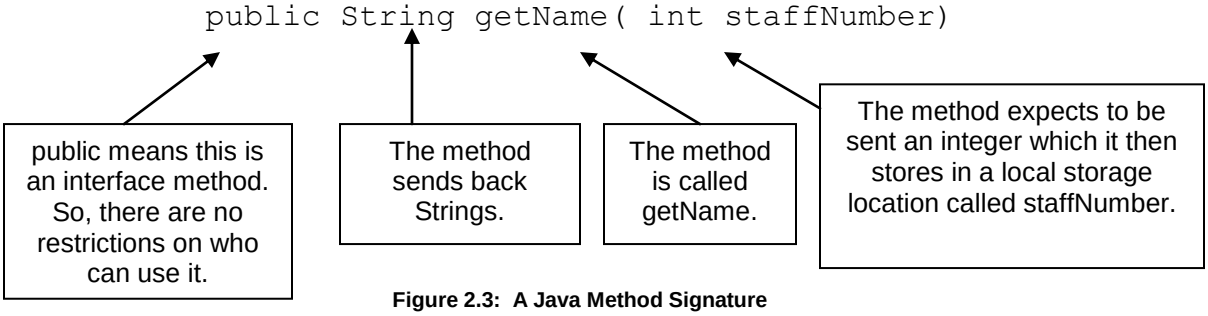


Figure 2.3: A Java Method Signature

You can guess what this Java method is supposed to do – reply with the name associated with a member of staff who has the proffered staff number[30]. The method will probably be a pure get accessor and will therefore not cause any change of state.

We should tell you that *formal Java* defines the signature as comprising only the method name and its parameters, not the access modifiers and return types[31]; but all of these aspects of the method are needed when you wish to use it.

[30] And, in Java, this name is likely to be a sequence of alphabetic characters – that is, a particular **String**.

[31] The reason is to do with polymorphism (see later) and how Java distinguished different methods with the same name – on the basis of their formal parameters but not their return types.

© OUDCE, January, 2013

Later in your career, you may come across the idea of an **Interface Definition Language (IDL)**. These crop up in a number of situations normally associated with distributive computing. IDLs typically express interfaces in terms of the function signatures that we have just covered, that is, they tell you *how* to use a facility, but not *what* the facility does. The information about functionality is normally provided in textual form, although this can sometimes be fairly hard to fathom!

Vendors group collections of software that achieve specific things – such as handling database manipulations or driving graphics hardware – into **application programming interfaces (APIs)**.

These often consist of little more than lists of available classes and methods together with information as to what they achieve and the related method signatures that permit you to use them.

Object-oriented systems

One of the advantages claimed for object-oriented software is that objects permit the development of computer systems which are conceptually closer to the real world situations that one seeks to represent[32].

For example, a Hospital system might have objects representing: Nurses, Patients, Employees, Visitors, Doctors, Beds, Wards, Administrators, Cleaners, Equipment, Operating Theatres, etc.

To discover how many free beds a given Ward has one might send a message to the Ward object (perhaps by clicking on its graphical representation). One might send other messages to decrease or increase the number of free beds.[33]

Developing a software system of this object-oriented type will involve defining the data characteristics, interface and methods of each object and then setting up the objects with the correct settings for their data. It will involve interpreting system functionality in terms of message passing and deciding on how the objects of the system associate with each other, because message transmission requires that objects have "knowledge" of one another and this sort of knowledge often translates into their having attributes that record the existence of the objects with whom they need to talk.

These steps must be taken. Don't worry if you are unclear about some of the terms – the list is for future reference!

- Discover how the system is supposed to interact with the outside world (use cases).
- Identify a plausible set of objects together with their attributes and methods.
- Determine how objects should relate to each other.
- Establish the interface of each object and any appropriate exception handling.
- Implement the objects and test them.
- Complete the construction of the system and test that.

These steps do not necessarily follow each other sequentially and one should expect some back-tracking and iteration.

The modelling techniques that we shall cover are designed to aid you in the processes of identifying the objects, their attributes and behaviour, and how they are related to each other; and to give you the notation to record your design decisions.

[32] We might observe that this is also one of the disadvantages! Nurse objects are not Nurses. Bringing your own behavioural prejudices to an O-O system is a recurrent danger.

[33] Perhaps we should say 'to increase or decrease our *record* of the number of beds' – see the previous footnote!

© OUDCE, January, 2013

The languages that are closely associated with the object-oriented view of things (which include C++, Java, C# and Smalltalk) provide facilities for defining objects and specifying their attributes and behaviour and how they relate to each other. They give few clues as to how you should design your system. For that reason, much of what we have to say is unrelated to the particular programming language you might use for implementation. Indeed, some of the modelling techniques we shall describe are applicable well outside of the ambit of object-oriented systems and might be applied to systems that are in no way object-oriented.

On the other hand, the object-oriented analysis, resulting as it does in a system specification involving objects and their relationships, tends to intrude on object-oriented design. Once you know how your system should look, in object terms, the detailing of software constructions to reflect that is perhaps easier than is the case in more traditional views of software construction.

A decision to construct a system according to an object-oriented viewpoint will in turn have a bearing not so much on the cost of construction of the system as in the proportion of that cost that is devoted to the different aspects of the development process – object-oriented systems may well involve you in spending more money in the early stages and less in the later ones.

Object-oriented systems are not necessarily terribly efficient at run time. It's probably true that there are generally more efficient ways to solve a software problem than those associated with object-oriented programming; but in many systems, efficiency is not a prime issue because sufficient processing power is available to conceal the considerable amount of extra processing and storage that object-orientation so often requires. Object-oriented languages are often conspicuously less efficient than low-level procedural languages and usually considered ill-suited to 'number-crunching' problems and some real-time applications[34].

Objects and distributed computing

Many modern computer systems are *distributed*: they involve networking and client-server processing. Such systems often work over the Internet.

Object-oriented programming is particularly suited to distributed computing because its central message-passing paradigm easily translates to a network environment.

Modern object-oriented message handling frequently involves communication between objects that are located on different processors. These may even be in different countries. In contrast, the standard procedure 'call' of imperative languages almost always involves a single processor and is local. Although modern procedural languages often provide generalisations of this, involving such ideas as 'remote procedure calls', these exist very much as a bolt-on to traditional views.

A second reason for the importance of languages such as C++ and Java in distributed computing is that they are languages that support **execution threads** – Java rather better than C++[35]. There is a considerable overhead to running a process in a CPU and much control data is needed. In C++ and Java, one can arrange a single process to subdivide into execution threads which, effectively, run concurrently.

Thus a web server can be 'simultaneously' handling multiple users in multiple threads. There is some small skill to thread programming but Java, in particular, makes it relatively straightforward.

[34] Champions of languages such as Java get very hot under the collar at statements of this sort – nevertheless, we believe them to be substantially correct, although we restrict them to considerations of execution. It can be argued that object-oriented systems are perhaps more efficient to *construct*.

[35] This sort of facility is more to do with the recent development of these languages than any innate feature of the object-oriented paradigm.

© OUDCE, January, 2013

In this way, servers written in Java do not clog up CPUs with multiple copies of their code – they load once, and spawn threads in response to user demand[36].

One more reason why objects are important to distributed applications is that object-oriented systems have a robustness that permits one to add new objects and behaviour to existing systems *as they run*. This allows distributed applications to evolve dynamically as changes are made, rather than having to be continually recompiled.

This robustness arises partly from the fact that object-oriented systems often postpone decisions about which method's code is to run until the point at which a message is actually sent. Traditional practice would settle the matter once and for all, at compilation. Flexibility of this sort comes at a cost to run-time efficiency, however, because run-time method resolution involves searches and data tables; and dynamism sometimes involves failure.

Technologies even exist to pass objects across networks – not just data, but behaviour too.[37] And individual objects may be stored in 'object-oriented' databases, to be resuscitated when processing demands it. You can imagine how this would translate to an implementation of, say, a shopping cart. We shall say a very little about these exciting developments in later chapters.

We choose Java in our coding examples because it is modern, object-oriented, free, and widely used. Java has a two-stage compilation process involving the abstraction of a 'virtual machine' which maps fairly directly to most modern architectures. This makes it particularly portable.[38] Java works on all modern systems and is, essentially, everywhere the same.

Classes and inheritance

Students often find it difficult to distinguish between objects and classes. Classes are software constructions that permit programmers to specify the behaviour and attributes of objects. The trouble is that classes often themselves possess behaviour and attributes themselves – that is, they closely resemble objects and in some languages classes actually *are* objects.

We hope to clear up these difficulties here.

Classes

Classes are **specifications** for objects. They **define** the attributes the objects have and their behaviour.

For example, in the Hospital system that we mentioned earlier, there may well be Nurse objects. Such objects could have attributes that record the name of the Nurse, and the Ward to which he or she has been assigned. They might also include behaviour that results in Nurses responding to a message requesting the Nurse's name with that information.

Here is how all this might be coded in Java. Don't worry about understanding all of the notation, though we expect you will be able to figure out a lot of it.

[36] In the first weekend in which the advanced diploma in computing at Oxford went "live" our servers fell over. They had been set to handle an upper limit of 50 simultaneous user threads, and we exceeded it - the first time this had happened.

[37] This is very similar to what happens when you obtain a web page – particularly one with JavaScript programming inside it.

[38] Don't worry if none of this makes much sense to you, but the point is sufficiently interesting to be worth a bit of investigation on your part, if you have the time.

© OUDCE, January, 2013

```
public class Nurse {

  String name;  // this defines a memory location
                //containing the Nurse's name.
  Ward ward;  // this defines a memory location containing
              //the Nurse's ward.

  public String getName() {
    //simply sends back the Nurse's name

        return name;

    }
}
```

Figure 2.4: A Class called Nurse in Java

This class definition specifies a class called Nurse. It stipulates that Nurse objects are to have a storage location called name, which should contain text (that is, a String) representing the name of the Nurse; and a storage location called ward, which indicates a Ward object that is to be associated with this particular Nurse.

The capitalisations are important and representative of what happens in object-oriented programming. We shall say more about them shortly – just note that **Ward** and **ward** would be two different things in Java.

The class also specifies a method called getName which is part of the interface that Nurses are to have.

A message to a Nurse object invoking this method might look like this.

```
thisNurse  .  getName();
```

Figure 2.5: Invoking the Get Accessor getName

You will remember that the word public in the line: `public String getName() {`

is how Java indicates that getName is part of the interface that Nurses have. It essentially says 'Anyone can use this method in messages.'

Instantiating Nurses

The code fragment in Figure 2.4 indicates how Java would enable a programmer to define a Nurse class. Nurse classes are *specifications* for Nurse objects. They are not themselves Nurses! If we want our system actually to possess some Nurse objects we shall have to create some.

This can be quite an involved process because real Nurse objects in a real Hospital system are likely to have a great many internal attributes. To keep things simple, we only showed two, but a real system might involve scores of them.

You will know that one creates data structures in traditional languages by a process known as **declaration**. One can do this in some object-oriented languages too, but the dynamic nature of these languages means that it is also useful to be able to create objects as and when one needs them, during the run-time life of the system.

© OUDCE, January, 2013

Steps associated with the creation of a new Nurse object

1. We must reserve sufficient space to contain the attributes that our class definition has defined Nurses as having.
2. We must populate these attributes with relevant initial data. For example, Nurses might have been defined to have an attribute called **name**, which holds character strings specifying the name of *this* particular Nurse. We should expect to enter the correct string into this attribute when we create the Nurse. Other attribute values would need to be set up too.
3. We must devise some way of referring to this particular Nurse object so that it can interact with the rest of the system (that is, so that other objects can send messages to it).

The process of creating a Nurse object (steps 1 and 2 above) is called **instantiation**. The Nurse object that one obtains is called an **instance** of the Nurse class.

Don't get confused about point 3 – it's about how the system refers to this particular Nurse. This has nothing to do with the internal attributes of Nurse objects and should not be confused with the 'name' attribute we mentioned earlier.

The simplified Nurse class definition in Figure 2.4 specifies that Nurse objects have *names* and *wards*. So creating a new Nurse object *instance* involves these things:

- allocating space in the system to hold the new object;
- using some of that space to record the class to which this new object belongs so that the system will be able to refer back to the class specification to determine what the object can or cannot do;
- setting the internal attributes of the new object in appropriate ways;
- setting up some way to refer to the new object (for example, referring to the Nurse object in Figure 2.4 as thisNurse).

Object-oriented languages permit you to create new objects by means of special interface methods called **constructors**[39].

Programmers devise appropriate constructors at the same time as they define the classes[40]. Nurse constructors are special methods in the interface that Nurses have – methods that accept data relating to the internal attributes of Nurses and store that data in the attribute locations specified by the class definition.

[39] There are formal technical distinctions between methods and constructors in Java but these are not relevant to us here and very many books on Java speak of constructor methods, which if technically incorrect, is often convenient.

[40] Java actually gives you a pre-written constructor for each class that you invent. Of course, Java is not able to decide on appropriate initial values for attributes so this may not be very useful. If you write your own constructors the pre-written one is withdrawn.

© OUDCE, January, 2013

Here's how instantiation looks in Java.

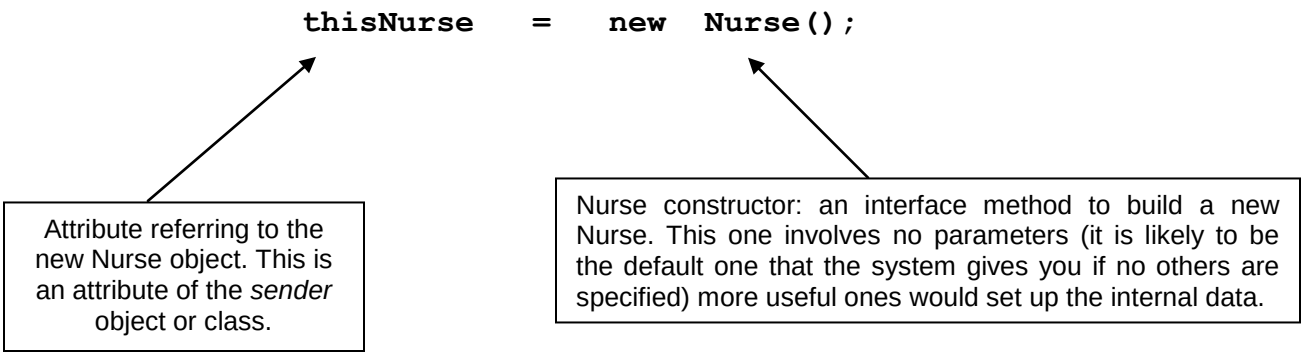


Figure 2.6: Instantiating a New Nurse in Java

The keyword **new** is how Java signals that a constructor is being called. It is the operator that causes space to be set aside.

A constructor method in Java **always** has the same name as its class – in this case, Nurse.

The messages associated with Java constructors are not usually thought of as returning values. All other Java messages do return values and we, perhaps loosely, tend to think of constructors as returning the *object* constructed. This is certainly not the case technically but does no harm for the moment.

One of the key features of an object-oriented system is its dynamism: the population of objects is likely to change as the system runs just as the population of Nurses in a hospital can be relied on to change. Constructors allow you to build objects dynamically, so they are rather more powerful than the static declarations of traditional programming languages, which tend to be resolved once and for all when the program is compiled[41].

Some object oriented supporting languages also have **destructors**. C++ does, for instance. These permit you to remove objects from the system in a controlled way. Java does not offer this possibility but Java systems do include a background "garbage collecting" facility that removes "unwanted" objects. In practical terms one can get dud objects swept up (see *referencing,* below).

[41] Of course imperative languages often do have facilities for the dynamic allocation of data space.

Object references

In Figure 2.6, **thisNurse** is a programming language variable that is permitted to refer only to objects that are Nurses. In Java, it might be declared like this.

```
Nurse thisNurse;  //variable referring to Nurses
```

Figure 2.7: Declaring a Variable of Type Nurse in Java

thisNurse might form part of the attributes defined by another class entirely and if it did it would then be situated within the specification for that class.

thisNurse is a **reference**[42] to Nurse objects[43]. The code in Figure 2.6 sets this reference to 'point to' (refer) to one particular Nurse: the one we were constructing. Later in the processing, the same reference might be used to refer to another Nurse entirely; but it must *always* refer to a Nurse object and not to some other sort of object.

Object referencing of this kind is very common in object-oriented programming and is supported by most languages associated with object-oriented ideas.

You might find this point clearer if, instead of the identifier "thisNurse", we imagine using the identifier "tonightsDutyNurse". You will see that that is a reference which is likely to change from day to day (unless some poor old Nurse has to work every night!).

You will remember that our simplified Nurse objects have an attribute called **ward** which can hold references to objects which are defined by a class called **Ward**.

You would probably not use constructors to define a Ward object at the same time in which you instantiated a new Nurse (in real terms, you would be more likely to allocate a Nurse to an existing Ward than to build a new one for him or her!). In all probability, the Nurse constructor that you would write in your Nurse class description would expect to be passed data that refers to an *existing* Ward. It is this data that would be copied to the ward attribute - data referring to a particular Ward object which, probably, had been long in existence.

You should now be completely happy with the distinction between classes and objects. The Nurse class explains the attributes and behaviour that nurses should have. Particular nurse objects are particular instances of this class and the running system may have many of them, each with their personal values in the attributes that the Nurse class description says they should personally have.

The destruction of objects in Java

Java manipulates objects by means of these references. There is no way to send a message to a Java object unless you have some way of referring to it, and one refers to objects by references.

Java garbage collection defines unwanted objects as objects that have no references. Such unreferenced objects are removed automatically from the system. Java garbage collection runs eternally as a background 'thread' in any Java system and it can be quite difficult to say when exactly an object might be collected as garbage. In contrast, a C++ programmer can destroy objects with greater certainty, by means of destructors which can include any appropriate tidy-up code.

---

[42] Although we should distinguish references from the objects to which they currently refer, in common with other writers we do occasionally blur the distinction, when it is convenient to do so and not particularly important. If you are unsure what a reference is, think of the name of a file and how it is from the file itself.

[43] The first word Nurse, here, specifies the **type** of object to which this reference can refer. It is Nurse, and so the reference can "reference" nothing else.

© OUDCE, January, 2013

Summary

1. One manipulates Java objects by means of references that are declared to have the same type as the class of the object to which they are going to refer.
2. A particular reference is likely to be the attribute of another Java object, or perhaps a (static[44]) attribute of a Java class.
3. References can identify different objects at different times as the system runs,[45] but the class of all such objects remains fixed[46].
4. You set the value of a reference by means of the assignment operator '='.
5. References may only ever refer to one particular object at any given time.
6. References are completely distinct from the attributes of the object to which they refer.
7. The story is very similar in most other object-oriented languages.

Vectors

The Hospital system that we have been discussing is likely to have a considerable number of distinct Nurse objects.

In point of fact, this system would be unlikely to handle Nurse objects by storing references in individual data items such as thisNurse, which is what we showed in Figure 2.6. It would be far more likely to use some sort of **collection class** of objects such as an array, or perhaps even a database.

Many data structures involve ensembles of things – lists, arrays, stacks, etc. Most object-oriented languages offer ready-made facilities to produce objects that correspond to different sorts of collection, and collections usually involve things of the same type – strings, integers, characters, etc.

Since objects are normally handled by reference in object-oriented languages, however, it's technically easy to implement heterogeneous collections, that is, collections involving objects from different classes.

A collection class object storing staff objects in our Hospital system might well need to hold Nurse, Doctor, Orderly and Admin staff objects. One could imagine calling it something like "surgical team" or "Accident and Emergency Shift".

Java **Vectors** can contain objects of different type (that is, defined by different classes). What the Vector actually contains is a sequence of addresses (references) to its objects. So the usual access problems associated with trying to manipulate collections of different types of things do not arise.[47]

[44] Explained later.
[45] In special cases one can set up references that are not allowed to change, and Java uses the keyword **final** to achieve this, but that does not concern us now!
[46] As always, the story is a bit more complicated due to inheritance, but this is substantially correct.
[47] In fact, there are some difficulties associated with Java Vectors. One has to **cast** objects back when accessing them (but this technical point is outside the scope of the present course). Furthermore, one cannot store Java primitives in Vectors. One has to resort to **wrapper classes**. So handling Vectors can be a little tricky until you get used to them. We shall use Vectors in some code that we show you in Chapter 8 but are not suggesting that we have given you a complete account here.

© OUDCE, January, 2013

Case sensitivity

Object-oriented programming languages are invariably case sensitive but the case conventions that have arisen in object orientation might be better described as case paranoid!

You may have been exasperated by our use of case up to now. We were following informal object-oriented conventions, which are as follows.

1. Capitalise the names of classes and use singulars rather than plurals – Nurse class, not Nurses class or nurses class.
2. Identifiers associated with specific objects start with lower-case letters – 'thisNurse'
3. Don't include white spaces in identifiers – 'thisNurse', not 'this Nurse'[48].
4. Where identifiers involve several words, capitalise the beginnings of all words after the first – 'thisNurse', not 'thisnurse'.[49]
5. Capitalise the first letter of class (static) attributes and methods.
6. Distinguish get and set accessors using the words 'get' and 'set'.

Some of these points may be mysterious to you, particularly point 5, since we have not yet covered class attributes and methods.

They will slowly begin to make more sense to you as your studies progress. We thought it would be helpful here to explain why most people working in object-oriented programming would suspect that Ward is a class whereas ward is a variable referring to a Ward object.

The conventions are certainly not hard and fast but they are respected surprisingly often.

If you insist on having classes that have identifiers starting with lower-case letters, or having object references that start with capital, then expect to have trouble when you show your code to other people. Like so many informal rules, you don't *have* to respect them, but you should have a good reason for flouting them.

Class attributes and behaviour

Objects are logically distinct from classes. Classes are repositories of information that specifies the characteristics (attributes) of their objects and how they behave (their methods and interface).

In many object-oriented languages, classes are able to store particular data that is important to *every* object defined from them, and also to encapsulate behaviour that is deemed to be important to *all* their objects. So, you see, classes can behave like objects.

In Java, one defines attributes and behaviour that relates to a class rather than its objects by means of the keyword **static**.

```
static double magnify = 1;      //used for scale change
```

Figure 2.8: Declaration of Class Attributes in Java

The above declaration relates to a data structure that each object in the class needed to access.

[48] It is quite common to use underscores, however – ward_Sister, head_Cook, etc.

[49] Of course, people tend to disagree about the constituent nouns in these compounds and might happily write playgroundArea or playGroundArea with complete insOuciance!

Class variables permit such data to be stored in a single location, rather than having to be duplicated with each object in the system. Static class attributes can therefore be used by every object of the class[50].

A good use of class variables occurs in the Java Math class. This class has a class variable called PI which contains a value for the mathematical constant π. PI is defined to be a double value (the highest precision value available) set to be as close to the real value of π as possible.

The Math class also has a static (class) method called **sin()**, which returns the trigonometric sine of a nominated angle.[51] In fact, most aspects of the Math class are static because the class essentially exists to provide mathematical services.

A class associated with screen windows might permit you to construct several Window objects (you can have several windows open at the same time) and might permit you to have methods that allow you to drag or resize windows. It is possible that each window would need to know the maximum screen size of your system and they could be told this if you declared a static, class, variable in the Window class, which recorded the information.

Instance variables and methods

Variables and methods relating to objects – in Java, the non-static ones – are sometimes called **instance variables** and **methods**.

Instance variables are storage locations that are reproduced with every instantiated object, so these variables are usually duplicated many times in the system and habitually contain different data.

Each Nurse object would have its own storage location called **name** (which is what one would want and expect). This is an instance variable because 'another one' is created with each *instance* (object) of the Nurse class.

Class attributes and methods are unique to the class but their services are available to all instances of the class and, quite often, to other objects and classes in the system. The Math class attribute PI is available to everything in the system and may be obtained by the expression Math.PI. The attribute PI is not duplicated in the system; it occurs only once and holds unique data.

Classes are not objects in Java but that information is of little use to you unless we become considerably more specific about what a Java object is. There are perfectly valid object-oriented languages, such as Smalltalk, in which classes are actually objects[52].

Part of the issue about classes and objects relates to a technical, but immensely useful, feature of many object oriented languages - their dynamism. In many situations the appropriate method to invoke when a message is sent is decided only at runtime. This decision is made by referring to the classes available in the system. In Java, this involves the classes **loaded** into the system - so if you wanted to use the sin function you would need to ensure that the Math class were available, that sort of thing.

[50] We leave it to you to figure out the possible security issue of one object changing a static value.

[51] Don't worry. This course requires no knowledge of trigonometry or π. These comments are illustrative only.

[52] The consequences of such purity can be further inanities – to add two numbers in Smalltalk you have to send one of them a message referring to the other, the reply is the addition. This involves a terrific amount of processing and adds to the runtime burden of Smalltalk systems.

© OUDCE, January, 2013

Classes are really present in running Java systems, and in many ways can be treated as special sorts of object. Further discussion of the technical distinction between classes and objects is outside the scope of this course.

We have now covered three sorts of variable in object-oriented programming in Java.

- Instance variables, which represent the attributes that objects are to have and which are declared *within* class definitions but *outside* method specifications.
- Class variables, which are like instance variables except that their declaration is preceded with the keyword **static.**
- Local variables (sometimes called *automatic variables*), which are declared within method specifications as aids to the algorithm that they represent.

One could perhaps include the parameters associated with method signatures as yet another sort of variable[53].

Inheritance

We invite you to reconsider the Hospital system that we sketched out earlier. We listed some of the Classes that might be expected in the system: Nurses, Patients, Employees, Visitors, Doctors, Beds, Wards, Administrators, Cleaners, Equipment.

An object-oriented analysis of the system would involve deciding on the classes and objects that the system should contain and how they should be related in terms of attributes, methods, messages and interfaces.

Some of the proposed classes have obvious commonalities but before we consider them, let's be more specific about what we mean by a 'Hospital system'.

We can hardly be completely specific here, because that would involve a detailed statement of system requirements. For the moment, let us loosely imagine that our system is an administrative one that is in some way intended to aid Hospital management.

Here are some of the attributes that we might expect of Nurse objects[54] in such a system.

1. name //holding the Nurse's name
2. ward //holding the Ward to which the Nurse has been assigned
3. payReferenceNumber //Payroll number
4. age //Age in whole years
5. address
6. telephoneNumber
7. hoursWorkedThisWeek //a cumulative figure used for pay calculations
8. holidayAllocation //the number of days of holiday left in this year
9. supervisor //a reference to another hospital employee
10. skillSet //a reference to a fairly complicated object whose attributes reflect possible skills

It's not hard to see that many of these would be attributes of Doctor objects too. Perhaps Doctors would not have supervisors or be allocated to Wards; but in general many of these attributes could apply to Doctors.

[53] Using "variable" in an informal sense, rather than however the Java programming language defines it.

[54] We are sure you could suggest others and might not agree with all of these. You may recognise this as part of the system's 'data dictionary' but that should also include constraint and typing information (age being between 18 and 65 and expressed in whole years, or whatever). Notice these start with lower case letters; you should be able to explain why that is.

© OUDCE, January, 2013

Actually, *some* of these attributes apply to all humans involved in the Hospital system. However, others are clearly quite specialised. For example, Patients would have symptoms or illnesses rather than skill-sets, and they would probably not qualify for holiday allocations!

Based on the above list of attributes, we might isolate these particular characteristics as applying to all humans[55] catered for by the Hospital system.

1. name //holding the name
2. age. //age in whole years
3. address
4. telephoneNumber

The choice we have made here is slightly arbitrary. We imagined that not everyone would be allocated to a particular Ward, for instance, and this does seem true, at least of Administrators; and you would not really want to define someone as not-human, simply because they had not telephone.

Let us imagine a class called Human. It would have the above attributes and basic behaviour related to them such as get accessors for each of these attributes and set accessors for most of them.[56]

Then Doctors, Nurses, Employees, and Patients are **kinds-of** Human[57]. When we come to specify their classes, we are going to need to repeat our definitions of these attributes in each case.

In fact, we can represent one view of how these classes relate by the following diagram, which is the first UML diagram of this unit.

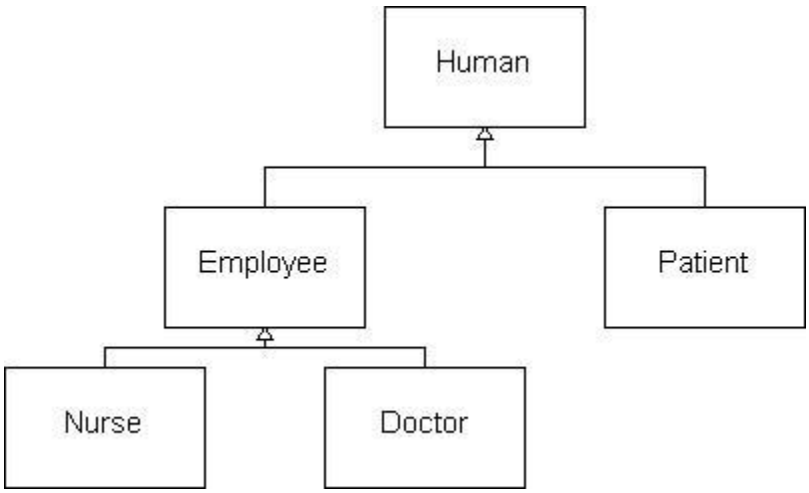


Figure 2.9: UML Diagram showing Class Relationships

Note the arrows in this diagram. They show that, in our view, Nurses and Doctors are kinds-of Employee, and Employees and Patients are kinds-of Human.

[55] We have not been capitalising 'humans' since they don't correspond to one of our posited classes. They are about to! Watch the case change!
[56] Imagine the debate about whether it might be necessary to change name occasionally!
[57] Based on the choice made, it seemed logical to imagine that Visitors, in this sense, were not kinds-of Human! On our definition, Humans have age. Would you really ask the visitors to your hospital their ages? Anyway, when we were in hospital not all our visitors were human!

© OUDCE, January, 2013

An alternative way to express this is to say that the Nurse and Doctor class are among the **subclasses** of the Employee class and that the Employee class and the Patient class are among the **subclasses** of the Human class.

Object-oriented languages usually offer facilities for subclassing of this sort. These facilities mean that attributes held in common, such as name, age, address, etc., can be specified[58] in a single appropriate place, such as the declaration for the Human class. They may then be **inherited** by the subclasses, that is, they do not need to be redefined subsequently in these subclasses.

In similar fashion, the Employee class might define these attributes:

1. payReferenceNumber //Payroll number
2. yearsOfService //How long they worked for us
3. hoursWorkedThisWeek //A cumulative figure used for pay calculations
4. holidayAllocation //The number of days of holiday left in this year

and, if it did, they would be inherited by the Nurse and Doctor subclasses, though the Human class would know nothing of them.

Here is how subclassing and inheritance are achieved in Java. We would define the Human class in the normal way, but when we came to the definition of the Employee class, we would type this.

```
public class Employee  extends Human {  //and the rest of
                                         //the definition goes here!
```

The instruction '**extends Human**' alerts Java to the fact that objects from the class being defined should have all the attributes and behaviour of objects from the Human class, plus such additional attributes and behaviour as the Employee class definition specifies.

Inheritance in analysis and design

Inheritance, in its most general sense, pervades object-oriented analysis and design. Again and again you will encounter techniques or concepts that emerge as a version of something else (**specialisation**). Again and again, you will encounter techniques or concepts that collect together a collection of things that are realised to posses strong commonalities (**aggregation**).

Summary

Inheritance is the process whereby one class is viewed as a subclass of another. The properties of the **superclass** then apply to the subclass but the subclass may have additional properties of its own.

[58] Yes, they are defined in a single place, but every instantiated object from the class or its subclasses will, of course, have their *own*, private copy of the variable.

Inheritance, references and derived classes

In Java, if class A extends class B, then A is a kind-of B and A objects are kinds-of B objects. Suppose you set up a reference that may refer to objects from the class B. You would do it like this:

```
B  myReference;  //declares a reference called myReference, which can refer to objects
                 //from B
```

You already know that myReference may only refer to objects that are from class B but perhaps you will not be surprised to learn that this reference can also be used to refer to objects from class A. The reason is simple, class A objects are deemed to be special sorts of class B objects[59].

Since quite a lot of the attributes and behaviour of class A is likely to be derived from the attributes and behaviour of class B, Java programmers would refer to class A as a derived class of class B.

Inheritance chains can get quite long, with A derived from B; B derived from C; C derived from D; and so on.

If your inheritance chains get very long it can actually be rather difficult to figure out how a derived class will behave; and the situation becomes more complicated in that object-oriented languages have different policies about which method should run. In some languages, this can depend on the class for which the reference variable was originally defined.[60]

Many experts on object-oriented programming recommend that one tries to keep inheritance chains reasonably short and that these chains should reflect the semantic meaning of the data.

Polymorphism

We hope that you are beginning to recognise that the key feature of object-oriented systems is their ability to model 'reality' very well. Most real systems have identifiable communicating 'objects' with attributes and behaviour and, we hope, may therefore be modelled fairly directly in object-oriented terms.

When you come to consider behaviour in abstract terms, you realise that very different objects can exhibit behaviour that is, in some sense, similar, but may practically involve different things.

To illustrate the point, an *Excel* spreadsheet, a *Word* document and a *PowerPoint* presentation are very different things in Microsoft *Office*, but they can all be 'printed', although the process involved will be different in every case. In object terms: each of these 'objects' knows how to respond to the 'print' message.

It is often convenient to have methods with the same name that do different things. In object-oriented languages, this facility is often called **polymorphism**.

You will recall that methods are specified within classes. What we are saying is that object-oriented languages, such as Java, permit you to specify distinct methods with **exactly the same name** in different classes.

What the method actually does will depend on your specification. If the method is an interface method in each of two classes, then the corresponding message can be sent to objects of each class, but its effects will be exactly as was specified in the corresponding code definition.

[59] This is not quite what we said originally about references, where we kept the story simple.

[60] We are thinking of C++ and the complicated story of virtual functions here. If you know nothing of C++, simply ignore this footnote.

© OUDCE, January, 2013

In this way, when you send a 'print' message to a Microsoft *Excel* spreadsheet, what happens is quite distinct from what happens when you send the same message to a Microsoft *Word* document.

Polymorphism applies to attributes too. Different classes can specify attributes with the same name but different meanings.

Of course, within a single class specification there will only ever be one method with a certain name, but the story becomes more complicated, because in Java, at least, methods are distinguished both by their names and by their parameters.

Two methods can have the same 'name'[61] but may be quite distinct, simply because they involve different parameters. You saw an example of that in Figures 2.4 and 2.5. Each related to a method called getName but the methods had different signatures. That means that Java can tell them apart and decides which one you mean on the basis of the data that you send with your message[62].

Method overriding

Figure 2.9 used a UML diagram to set out how some classes were related. You will recall seeing the arrows in that diagram. Oddly, the arrows pointed *upwards* from subclasses to classes. We are now going to explain just why that is.

When a message is sent to an object in an object oriented system, the class definition must be accessed in order to decide which code snippet to run. As we have just observed, this can involve sorting out several different methods with the same name but different parameters in order to establish which one applies to this particular case.

Quite often, however, the class definition will not have a method corresponding to the message. This happens when the class is a subclass of another one, and inherits the method in question from a superclass.

If the system cannot locate the method in the current class, it has to go to the specification for the superclass and try to find it there. If it cannot find it there, it must go to the superclass's superclass, and so on. In each case, the system needs to go from a subclass to a superclass, not the other way round. That is why the arrows point as they do in UML diagrams such as Figure 2.9.

There is, however, another aspect to this process. It may well be that that superclass *does* have a method with the same identifier as the one in question, but that the subclass has one too!

Object-oriented systems allow one to change one's mind about what a method should do in a subclass. You do that by simply writing a new method in the subclass with the same identifier as the method in the superclass. This process is called **overriding**. Because the search proceeds upwards from the subclass, the subclass method is found first and the search stops, never getting to the superclass[63].

[61] We should start using the term 'identifier' here rather than name.
[62] By matching up with the expected parameters. Remember that return types are not used in making this distinction, which is why Java does not regard them as part of the method signature.
[63] And, yes, object-oriented languages have mechanisms to get at the overridden method too. It gets complicated.

© OUDCE, January, 2013

Overriding is a strong reason for keeping your inheritance chains short. It can be extraordinarily difficult to determine what will actually happen when you have multiple overridden methods, and the Java story is probably rather clearer than what happens in C++.[64]

Multiple inheritance

The search to resolve a method specification that we have just described is fairly close to what actually happens in languages such as Smalltalk or Java.[65] The search described assumes one thing very clearly – that a class can only have one superclass!

There are languages, including C++, in which this is untrue. The story then becomes considerably more complicated because classes can inherit from multiple classes.

Although the cliché used to be that 'multiple inheritance' was a very bad thing, this is certainly not the common view today. We have even heard people complaining that one of the problems with Java was that it did not permit true multiple inheritance![66]

Multiple inheritance is particularly useful for event programming and for programming using the fashionable 'Model View Controller' (MVC) design pattern. We shall return to MVC later, but we shall have no use for multiple inheritance in this course.

Overall summary

We have now covered the following object-oriented ideas.

1. Objects, attributes, behaviour.
2. State and state-related behaviour.
3. Messages, parameters, return values.
4. Constructors and referencing.
5. Classes, class attributes and class behaviour.
6. Inheritance, subclasses, superclasses, polymorphism.
7. Method overriding.
8. Object oriented systems.
9. Object oriented languages.

You should make sure you have some understanding of them– these ideas are easier to use than to explain and you will find them progressively more reasonable as you work on them.

[64] See our previous footnote on virtual functions in C++.

[65] Just out of pure interest, Java owes quite a debt to Smalltalk, and the people who first designed Java were accomplished Smalltalk programmers. Java bytecode is identical to Smalltalk bytecode (not a lot of people know that!).

[66] Java interfaces are often thought of as devices to permit a sort of multiple inheritance.

© OUDCE, January, 2013

Exercises

1. Do all the objects in an object-oriented system have to represent living things?
2. Open Microsoft *Word* and start a new document. Type a single line of text, such as 'The quick brown fox jumped over the lazy dog.' Then key F10 and F -. What happens? (You should find that the letter F is not echoed in your document.) Explain this in terms of state-related behaviour.
3. The object-oriented system, very partially described in 2.9, has the requirement that one must be able to determine which doctor is treating a particular patient. This will involve some system objects having particular attributes and behaviour. Imagine that there is an Admin class and that an object from that class finds the information by sending a message to a patient object. Relate that to the attributes and methods of classes in the system.
4. You may have encountered the old concept of entity-relationship modelling. Write some notes relating this to object-oriented modelling and spelling out any differences that come to mind.
5. Suggest a class attribute for the Admin class (see Question 3) in the Hospital system. (One such attribute might be the VAT rate.)
6. We have not yet begun to address object-oriented analysis and design but if you have previous experience of other methodologies, such as SSADM, then try to suggest aspects of object-oriented design that will be similar to these, and aspects of it that are likely to be quite distinct.
7. You are designing an object-oriented system from scratch. List the things that you will need to specify.
8. Although we have said that the object-oriented view of things is particularly useful in distributed systems, we have not touched on what makes Java, in particular, so portable. Use the Web to investigate Java bytecode, compilation and interpretation. (This exercise may involve more time than you have and is not directly related to the subject matter of this chapter but is a useful thing to do.)
9. Explain how one could model a website in object-oriented terms. The pages of the site are to be the objects. You will have to come up with a reasonable view of what a message might be and how it would be sent.
10. Object-oriented systems, such as the Hospital system that we discussed earlier, are often said to model real-life situations very well. There are problems with that because people can forget that the systems deal with convenient human abstractions rather than real things. Construct an example to illustrate this point.
11. Spend a little time searching the web and investigating object-oriented ideas. Skim read most of what you find because the idea is to get you familiar with the terminology. If these ideas are genuinely new to you, discuss them with your colleagues and make some notes yourself outlining the key ideas. New ideas take some digestion but do eventually become familiar.

© OUDCE, January, 2013

Chapter 3: UML Diagrams

Introduction

This chapter focuses on the structure of systems and introduces you to some diagrammatic techniques which may be used to model key aspects of them. The techniques can be used either to provide an account of an existing system, or as part of the process of specifying a new one, but their primary rôle is to communicate technical information to humans.

The diagrammatic techniques are based on those of the Unified Modelling Language (UML). This rich, pictorial language has many features that are not immediately useful to us. You should be warned, right from the outset, that we do not intend to cover UML in full generality.[67]

UML has been used for a wide variety of purposes and has proved useful in areas that are at some remove from those for which it was originally designed; many of these, quite far from object oriented ideas. Our main stress will be on object oriented systems but we shall occasionally look at wider applications of UML and in this chapter at least, the discussion is nearly always applicable to any system at all.

Diagrams and modelling

It is generally believed that a complicated software problem is best viewed from a set of independent vantages.

One might specify:

- what happens (the functional viewpoint);
- what data is processed and stored (the data-driven viewpoint);
- who interacts with the system (the user-driven viewpoint);
- how and when things are to happen (the non-functional constraint viewpoint).

Obviously, these considerations will occur in almost any system. What is in question is whether it is advantageous to consider them separately. Every view you take of a situation is at the expense of another you might have taken, but is it effective to analyse function and data separately at first? Your delivered system will have to handle both properly, and the truth is that there are no such things as 'independent vantage-points'.

> Software models focus on particular aspects of a situation at the expense of others: they both reveal and conceal.

Viewing separate aspects of a problem in isolation will only be useful if the chance of merging one's ideas into a uniform, consistent solution is likely to be good.

Most approaches to systems analysis and design do involve this separation of viewpoints. Indeed, the word 'analysis' partly means 'breaking up into smaller issues'. You should note, however, that a particular method might be more applicable in some areas than others. The techniques that one chooses depend on a number of things, of which the nature of the problem to be solved is the most important, with staff expertise and tool availability also to be considered.

[67] In point of fact, UML is so rich that few of its users would exploit every aspect of it.

© OUDCE, January, 2013

It's obviously possible to specify a system entirely without the use of diagrams,[68] but it is very hard to use complicated linguistic or notational specifications as the basis for intelligent discussion involving teams of people. People are likely to forget, misunderstand or misinterpret; and they will be forced to express themselves in their own private shorthands. It will also prove very hard to record the result of their debate clearly.[69]

So, diagrams have proved recurrently useful in software engineering. The trick is to find simple diagrams that reflect important ideas that are sufficiently independent to make using them useful and productive.[70]

UML

UML arose as part of collaboration between leading authorities on the construction of object-oriented software. They worked by adopting what they considered to be the best aspects of various diagrammatic techniques that predated it. It first appeared in 1995 and has been developed and augmented since then.

We are covering UML because it is so commonly used and because people have consistently found it to be useful and appropriate.

UML is primarily an analysis and development tool. It does some things very well, but a system developer will need more than UML diagrams to specify such things as code logic, non-functional constraints and performance; and that's saying nothing about testing.

UML is used to produce diagrams that model aspects of systems. This would be as part of a **software development process**. You will probably already know about various views of how software should be constructed, and we have nothing to add to that here other than to observe that UML diagrams may arise as products of a development process but have nothing to do with the nature and organisation of that process.

UML is designed to be independent both of the development process adopted and of the programming languages to be used.[71] On the other hand, development methodologies have been invented with UML in mind: the **Unified Software Development Process** is one such methodology and there are others.

One key advantage of UML is that it is widely understood and used. Ideas expressed with UML diagrams are likely to be directly intelligible to a considerable range of people. The language has become part of the skill-set of any modern computer professional and there is a range of sophisticated support products that make drawing the diagrams easy and permit such things as code generation.

Violet

Although we decided not to ask you to acquire a UML tool for this unit, we are using an open source UML editor called *Violet* to produce our own UML diagrams.

[68] One could argue that the program code of the system does that!

[69] The jaundiced reader might like to consider the process of minute-taking that is so standard in meetings. These are usually non-technical and (allegedly) in plain English! Have you ever read minutes and wondered whether they were for the meeting that you attended?

[70] Remember our stress on the fact that models *simplify*. This can cause contradictions between different models. Our set of modelling tools will have to reduce the chance of that happening as much as possible.

[71] This statement needs flavouring with a pinch of salt!

© OUDCE, January, 2013

Violet is very easy to use and you might like to download it. The UML diagrams in this text were produced using *Violet*, saved as jpeg files, and then embedded them within *Word* documents. You may download *Violet* from: http://www.horstmann.com/violet/.

There are other open source UML editors around, such as ArgoUML. There are also a number of professional packages for drawing UML, of which Rational Rose is the most famous.

Violet is attractive because it is extremely simple and very portable. It is not intended as an industrial tool and it cannot handle some kinds of UML diagram in its present form.

Java programming involves a two-stage compilation process. The first stage produces Java class files in a low-level language known as bytecode. *Violet* is delivered as a set of these class files. It is very likely that your existing system will be able to run them as they stand since it is almost certain that your system has a Java virtual machine within it.

It is, however, possible that your version of Java is not sophisticated enough for *Violet*, in which case, you will need to download a more up-to-date version of the Java run-time environment.

We had to do this ourselves, and downloaded a self-installing environment appropriate to our Windows system. We had absolutely no problems with downloading and running it and we would not expect you to have any either.

We run *Viole*t by using a Windows batch file which we keep in the same folder as the downloaded *Violet* jar file. Our batch file has the single line:

```
java      -jar      violet-0.15-alpha.jar
```

(The name of the jar file was the name of the downloaded file.)

We simply click on the batch file and everything works; but it does depend on Java being installed correctly on the system[72].

Syntax

We must warn you that UML diagrams have a prescribed syntax. They normally consist of lines, shapes, arrows and text. Different kinds of arrow mean different things and you need to get that right.

If you use *Violet*, these decisions are made for you by the software package. If you use another method for producing diagrams you will have to ensure that you conform to the UML specification, which is readily available via the Internet.

UML is so widely understood that you must take pains to use it correctly or risk looking amateurish.

[72] The most common problem relates to whether a path to the Java system is set correctly. If you have installed Java correctly but still can't run *Violet*, it is likely that that is the problem. There are web sites that will test whether you have Java installed correctly. Go to http://www.java.com/en/download/help/testvm.xml if you are unsure about this.

The UML diagrams

We shall discuss the following sorts of diagram.

- Use case diagrams.–These are used to describe users' interaction with the system: *what* happens, rather than *how*. These diagrams specify system capability.[73]
- Class diagrams.–These are used to describe the classes the system contains and how they are related. This relates to the *static* nature of the system but it includes information on the possible *dynamic* interactions between objects; because objects can only send messages to one another if they know about one another, and their knowledge of one another often translates into *associations* between classes.
- Behaviour diagrams. These subdivide into:
    - ❖ State diagrams, which are used to represent the internal state changes of objects
    - ❖ Activity diagrams, which are used to represent some sort of control flow, and often directly related to a particular use case
    - ❖ Sequence diagrams, which are used to represent the sequence of interactions between objects associated with some aspect of system functionality

Your interest in these diagrams is first with a view to understanding them and second with a view to being able to draw them yourself as part of a software development process.

We shall introduce the diagrams to you descriptively, as part of an account of a supposed existing system that will form a case study. We hope that you will learn to use and understand these diagrams by seeing how we use them. Once you have some understanding of the technique, you should then actively try to see how other people use it; and you can find a wealth of information on using UML on the Web.

Most software jobs involve the maintenance or extension of an existing system[74]. One might well begin such jobs by constructing models of what is in place; and proceed by extending those models as part of the analysis and design of the required solution. One sometimes even inherits models as part of a system's documentation. If would be extraordinarily unprofessional simply to ignore one set of models and construct another set; and doing so might even be something that was expressly forbidden by the contract[75].

The system with which we shall be working is one that you may have seen. You can find an account of this system at the Porterhouse Library.

That document is described as a requirements definition for the Porterhouse Library's existing system – a system which does not yet involve computers. The understanding is that a computer system is wanted; consequently, a detailed description of the existing system provides a sort of specification for the computer system that will replace it.

In real life this description would be the subject of a period of intense **requirements elicitation** and **analysis**. That process would result in a more technical specification of the system, which would form the basis of any contract to commission it. These processes are important and you may well have already had some exposure to them. We shall not treat them here.

---

[73] We have to tell you that there is a certain snobbery about the pronunciation of 'use case'. It should be pronounced 'Yews case', not 'Yoose case'. My old, Irish, colleague, Raymond Flood, used to complain bitterly that in his dialect of English the two had no distinction!

[74] One of the objections we have to many established texts is that they seem unduly geared to the production of systems *ab initio* – which is not what you will experience in your first software job.

[75] Though, sad experience leads us to tell you that the associated documentation is not the system itself and you have to be sure that a model you inherit is really accurate.

© OUDCE, January, 2013

We wish to use this system description to illustrate the use of UML, and in that work you may imagine *either* that an object-oriented implementation of this system exists and that we are merely describing it, *or* that our diagrams simply describe the system in object-oriented terms and are therefore a specification for an object-oriented implementation.[76] We shall not give a complete account of this system but just illustrate UML by looking at a few of its features.

[76] This dual view of what we are about will cause confusion, and resolving this is an integral part of our exposition.

© OUDCE, January, 2013

Chapter 4: Use Case Diagrams

What a system actually does is clearly the most important thing about it; the UML approach to identifying that involves **use cases**.

Use cases are identifiable bits of behaviour. We describe them by means of **use case diagrams** and **use case descriptions**.

Use case diagrams

Use case diagrams:

1. identify a specific use of the system, that is, something the system does or should do;
2. identify external entities, called **actors,** that use the system for services;
3. loosely identify the boundary of the system (scoping);
4. give indications about the placement and construction of system interfaces, particularly human-computer interfaces;
5. can be used to generate test specifications.

Amplifying points 1 and 3: use cases correspond to external uses of the system. Actors are externals, whereas use cases describe the facilities provided for them.

Actors are people or things that interact with the system without being a part of it. They may include categories of users, scientific sensors, or other computer systems and they should achieve something as a result of their participation in a use case. Actors are not passive, we used the term ”interact” above and we meant it because Actors should exhibit behaviour of their own.

Actors are entirely characterised by their roles. They are not the same thing as individual people.

A particular person might use a banking system both as an ordinary customer (having an account and needing account services) and as a privileged user (maintaining the system and setting its capabilities). We humans would recognise the particular individual in both situations; but the system would have no such discernment. So far as the system is concerned an actor is entirely characterised by rôle. The distinct activities of our individual appear to the system as involving completely distinct actors, who might be called: *customer*, *system_manager*, *accounts_manager*, etc.

It is important that you understand this, because use cases will make little sense to you otherwise.

We warn you that use case diagrams appear so incredibly simple that you might be tempted to disregard them completely. Nevertheless, they are probably the most important of the UML diagrams – not for their complexity or detail, but because they set out clearly what the system does, where its boundaries lie, and what kind of things interact with it.

Use case descriptions

Use case diagrams would normally be augmented by **use case descriptions**. These give a textual account of the use case, one that might perhaps include other UML diagrams. They would also record any non-functional requirements associated with the use case.

One can often make a start at identifying possible use cases by looking at the verbs used in the system specification, which in our case concerns the Porterhouse Library[77].

[77] We have mixed feelings about this sort of advice. If you don’t know where to start, it at least gets you going.

© OUDCE, January, 2013

From that, we see that two activities that could occur in the Porterhouse Library system are borrowing or returning books. Here is a use case diagram for these activities.

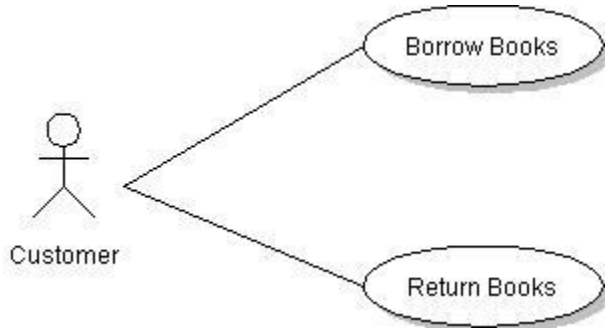


Figure 4.1: Use Case Diagrams for Borrowing and Returning Books

Syntax notes

1. Each actor is represented as a matchstick figure.
2. Each actor is connected to the use case by a line.
3. Each use case is represented by an ellipse.
4. The connecting line can end in an arrow-head to indicate that the actor *initiates* the use case if the actor does indeed do that. We could have used arrows in Figure 4.1.
5. The actor is given a name which is based on this particular role and not on such things as job titles or personal names.
6. A single actor may be involved in several use cases[78].
7. A single use case may be involved with several actors.
8. A single diagram may contain several use cases and actors.
9. Indeed, one can collect all of the system's use cases into a single functional description of the system, provided that this is not too complex. If one does that, it is often useful to represent the system boundary by a rectangular box, with the actors *outside* and the use cases *inside*.
10. Use case names often involve one noun and one verb (as in the above examples). They should certainly not involve conditions or decisions.
11. *Violet* permits you to attach explanatory notes to diagram elements. We have not done that here.
12. It is standard practice in UML to represent the system boundary in these diagrams. We have not done so here but if you wished to do that you would show it as a box that kept the ovals inside and the matchstick figure(s) outside.

We have actually overcomplicated this diagram because it represents two use cases. We combined them because they both involve the same actor 'Customer' but it might have been better to use separate diagrams.

The diagram is simplicity itself, but what is being described may not be. Here are some things to consider, and we suggest that you read and consider them carefully because they include some of the most important points that we shall make.

[78] Deciding whether or not to do this is sometimes a matter of clarity. In our example the Customer actor could have been subdivided into Borrower and Returner; but this seems to be unnecessary precision.

© OUDCE, January, 2013

Issues arising from Figure 4.1

- How is Customer defined? We know from the library description that there are actually categories of library users who might undertake the activities represented. More than that, these categories of user are supposed to have different capabilities with respect to the number of books they can borrow and how long they can keep them. So should Customer be further split into categories, or should we, perhaps, start thinking of subclasses even at this early point?
- How are the use cases defined? We called one of them 'Borrow Books'. Is this supposed to mean the act of successfully removing books, on demand, from the library? Or, more likely, is it supposed to mean the transactions associated with an *attempt* to do so – transactions that may well *fail* to result in books actually being removed?
- People making their first attempt to model an object-oriented system have a tendency to pluralise too quickly. The actors shown in Figure 4.1 are singular – Customer, not Customers – but the use case titles involve plurals, speaking, as they do, of Books. This may be reasonable, but close analysis of the situation might well find that what one has to treat is an attempt to borrow a single volume or book. In most cases, you will find it best to attempt consciously to keep things singular at first, only admitting plurals when absolutely necessary[79]. So the use cases might be better named Borrow Book and Return Book.
- The moral is 'Resist plurals in the early stages of analysis[80].'
- On a semantic note, if we call the use case 'Borrow Books' but the associated activities may not result in books being borrowed, are we setting ourselves up for misunderstandings and possible errors?
- On the other hand, since the system requirements definition does speak of borrowing and returning books, is it more important to relate our UML descriptions fairly closely to the textual system description, and to represent the concealed complexity of these use cases with other diagrams that qualify them?[81]
- Do 'customers'[82] actually 'borrow books' in the sense of using system facilities? This question is a tricky one and the response depends on exactly what it is that we are modelling. If there is an existing computer system that we seek to represent using UML diagrams (and that is one of the positions we are taking), then the question arises: 'Which actor "borrows books"?' Is it not likely that a member of the library staff will in fact interact with the system being modelled? In all probability, what will happen is that a member of the public makes a verbal request to a librarian on the issue desk and it is the librarian who interacts with the system to 'borrow books'. In terms of the *system* being modelled, the use case Borrow Books involves an actor whom we might name Librarian. On the other hand, if we are using UML diagrams to describe an existing, non-computer system, it might be appropriate to talk of customers 'borrowing books'. This will probably still involve the good offices of a member of the library staff, but the staff member is now thought of as part of the system, and not an actor as such.

The above points are intended to indicate the sort of ambiguities and simplifications that any system modelling process entails. These problems are inherent to the analysis of systems. Similar problems would arise with any technique that we could show you.

[79] This is exceptionally good advice.
[80] Almost any time you think they are essential, you are wrong!
[81] You will see shortly that Borrow Books is important. It is the main success scenario (q.v), so it might validly be used as the name for this use case.
[82] In this paragraph, at least, we shall distinguish informal descriptions of actions from defined use cases by means of inverted commas.

© OUDCE, January, 2013

The fact that they occur here and are so clearly visible is not an indication of any failure of UML, or even of our use of it! On the contrary, it indicates the difficulties we always have when we seek to represent functional ideas and how this simple technique can be used to bring them out.

The point is to expect these difficulties and to resolve them by better and better diagrams. Wherever you start, an initial simplification will be needed, and you can hope to do better through an iterative process of revision and amplification.

You should be aware, however, that total clarity and precision in these descriptions is something you read about in textbooks and never see in practice! We encourage you to keep a record of your own unease with the story we tell, because no account will ever be completely satisfactory.[83] More than that, informal notes of worries that you may have in these descriptions are a vital part of the modelling process and might reasonably be added to the use case descriptions. You should get into the habit of not placing all your eggs in one small basket!

At this point we ask you to agree that one could produce a series of diagrams like Figure 4.1 covering every aspect of how the library system is used and identifying categories of user of that system.

The diagrams would simplify things; but any two people working with you on the library system would have a fairly clear idea what the system does, whom it involves, and what it does not do. If someone on your 'team' asked 'Does this system handle stock control?' you could say '"Can't you read? Is there a use case diagram describing that?' If someone else mentioned that the original system specification said that such and such a thing happened, you should be confident that you could identify the use case (or use cases) where this occurred.

Further precision

We know that the Borrow Books use case is probably misnamed. It is likely that the activities we have in mind for this do not always result in books being borrowed, so this name is likely to cause continuing confusion. The likelihood of this happening may be small, but why take the chance? It's far better to give the use case a less misleading name.

And, of course, the use case splits into a number of possible **scenarios**, which might otherwise be treated as use cases in their own right. We could list these as follows.

4.1 The customer attempts to borrow books but is found not to be registered with the library.
4.2 The customer attempts to borrow books but is found to have already borrowed so many books that he or she may not withdraw the number that is attempted.
4.3 The customer attempts to borrow books but some or all of them are found to be subject to some sort of prior request.
4.4 The customer attempts to borrow books but is found to already owe the library money for fines or to have outstanding overdue books.
4.5 The customer attempts to borrow books but the library is not currently lending them, due to annual stocktaking.
4.6 The customer attempts to borrow books and ……..*succeeds*!

Scenarios

A **scenario** represents one possible instance of a use case. Scenarios present us with a finer-grained analysis of what happens. You can see that Borrow Books splits into at least six scenarios, with some more likely to occur than others. We saw something similar when discussing the Customer role. That could be split into sub-roles giving more precision to the activities that concern us.

[83] And after all, we are attempting a description of an existing system that is a bit simpler than the system itself. Of course we must simplify and occasionally misrepresent.

This kind of iterating subdivision is characteristic of object-oriented systems and corresponds both to ideas of inheritance and to ideas of classes and the instantiation of objects.

One might regard a use case as a *class* of system behaviours, and scenarios as objects instantiated from use case classes. You will see below that this analogy extends diagrammatically even to ideas of inheritance and containment.

Each of the above six scenarios is 'a-kind-of' Borrow Books in that it represents one possible instance of it. Should we model them, then, as six different use-cases?

The answer depends on the system and the recurrent judgement that one makes when modelling entities: is something best regarded as an entity in its own right, or as an aspect or attribute of another entity?

In practice, one simply makes a decision based on the existing situation, which includes, of course, the specification documents and their terminology. As with all such decisions, one is more likely to detect evidence that one is wrong than confirmation that one is right!

We seek to manage complexity by providing an economical account of a situation. In this particular case we shall regard the six activities listed above as *aspects* of the Borrow Books use case, that is, scenarios. This decision comes, however, with the proviso that we consider coming up with a better name for the use case itself, because five of the six scenarios above result in no books being borrowed at all!

And yet, some of the above scenarios are more likely to occur than others. One might claim that Scenario 4.6 occurs 80% of the time, while scenario 4.5 probably occurs in less than 1% of the occasions that a Customer attempts to borrow books.

We describe Scenario 4.6 as the **main success scenario**,[84] and this will be the scenario that occupies us in our initial modelling.

As a rule of thumb, you might expect that your use case has one clear success scenario and that this is likely to be closely related to how you secretly think of the use case.[85] If that is not the case, you might like to re-examine your analysis and decide whether you might be better advised to split your use case into two separate ones; but no rule of software analysis is hard and fast[86].

The other scenarios are, in some sense, boundary or error conditions. These are the ones that occupy you in 80 per cent of your work, but actually may occur less than 20 per cent of the time (the '80/20 rule').[87]

A system succeeds or fails in its boundary regions and yet there is a limit to how much error processing you can build in because it comes at the expense of fearful complexity. It may be that successively exhaustive exception processing can be the stuff of subsequent deliveries of the system, but here, of course, we are not involved in design, only description.

[84] Some authors insist on describing this as the 'happy day scenario'!
[85] And in this case, how we initially named it!
[86] Except for this one! As always, you don't have to follow "rules" but you do have to have a reason for not doing so!
[87] This is sometimes called the Pareto rule, after the nineteenth-century Italian economist who observed that 80 per cent of the country's wealth was owned by 20 per cent of its people. Would anyone like to guess whether this has changed?

© OUDCE, January, 2013

An important feature characterising scenarios is that they should achieve or result in something tangible relating to the actor involved. Use cases describe system functionality from the perspective of a user; scenarios are the possible outcomes of use cases.

Further elaborations

One use case may be recognised as always involved in the execution of another one. For example, the library desk may undertake an activity called Locate Account. This could involve an actor called Librarian in interrogating the system in order to locate the account details of Customer. Here is the use case diagram.

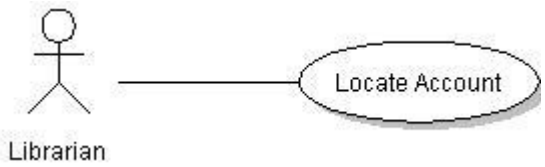


Figure 4.2: The Locate Account Use Case

This use case is recognisably part of the Borrow Books use case but, following the points we made about actors in the discussion after Figure 4.1, we rename the Customer actor as Librarian and represent the connection between these two use cases in this diagram.

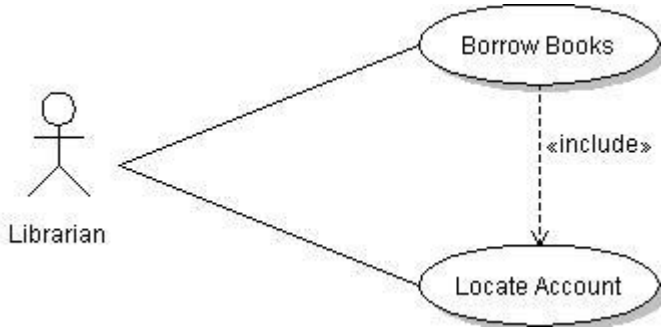


Figure 4.3: One Use Case Included in Another

Figure 4.3 explains that, whenever the Borrow Books use case occurs, the Locate Account use case also occurs, as part of it. Note the different arrow-head used for the include relation. Note also that it is possible to imagine that Locate Account will occur in other circumstances within the system and therefore be associated with other use cases.

Once again, this raises semantic problems. You will remember from Scenario 4.1 above that it is possible to imagine someone attempting to borrow books without being a member of the library. What we have just said, about use case inclusion, argues that Locate Account is always performed, presumably giving a negative result in this scenario. On the other hand, the system being modelled may involve non-registered customers being registered at this point. If this is the case, one would expect to have a better name for the use case in question; since Locate Account seems to us to imply that the account does in fact exist. Perhaps the name Get Account is less pejorative[88].

[88] When you first start modelling systems you tend to be irritated by issues associated with the semantic connotations of names for your team. However, after many arid debates arising from this, you may well resolve to obviate problems of this sort right from the word go by clever choice of terms.

© OUDCE, January, 2013

You should be able to see how complicated even this initial modelling is likely to become. It needs multiple attempts and considerable reflection, plus a very clear specification. It is likely to involve a permanent need to clarify what happens or is wanted with the system commissioners, and you will realise that in real-life situations you are likely to find that no one knows clearly or that different people have different views. From the specification we have, can *you* tell what happens when a customer is not registered with the library?[89]

Quite generally, these, and the other UML diagrams that we shall show you, have a great deal of meaning within them and need close, dynamic examination. When you produce a diagram, get into the habit of considering its semantic consequences: you may not be able to resolve them at once, but they must be noted and addressed sooner or later. In particular, watch out for consequences that might preclude any activities that are specified as requirements!

You will probably find that much of your analysis at this point is linguistic. If you are working in teams you will need to ensure that everyone is using terms consistently and in the same way.

Notational conventions

We have now shown you quite a lot about use cases but there is considerably more to be said, and there are further techniques involving such things as **extension** and **generalisation**. You will find that the *Violet* tool does cater for these ideas.

We shall not use them in this course. By all means find out about them yourself, but don't use them, either in our exercises or assignments. UML diagrams can contain a welter of constructions but their primary use is communication. Like many software companies, we shall place restrictions on what you are permitted to use. In that way, we can ensure that we are all speaking the same diagrammatic language.

> You are only permitted to use the notational techniques that we show you. If you bring in further notations you may well lose marks in assignments and projects, even if you are using them accurately! This is our "company standard".

We have spent some time discussing the use cases associated with issuing books on loan, and since we need to give you some experience of use case descriptions, we now offer one for the 'Issue Loan' use case. The first thing we should note is that we have not supplied you with a use case diagram for that. We ask you to imagine that we have now resolved the debate about who borrows books by deciding, following the system requirements definition, that the use case we need involves an actor called 'Librarian' and should be called 'Issue Loan'. You should be able to draw the use case diagram yourself but, as you will see, it involves other use cases.

When one writes a use case description, one aims for clarity and economy. Descriptions of this kind are often tabulated or at least structured (that is - involving numbered points). One can hardly answer the question "How much is enough?", but one should also be mindful that massive complexity defeats the point of this sort of modelling.

[89] Once again, this aims to be a practical course – the experts will tell you how specification clarity is to be achieved. It rarely works quite as they suggest.

Use case description for 'Issue Loan'

1. This use case involves the actor 'Librarian'.
2. This use case occurs when someone asks a member of the library staff (Librarian) to issue books.
3. The **pre-condition** is composed of an individual who is registered with the library, and a collection of book titles.
4. The Librarian first logs into the system. (This involves a password.)
5. The Librarian identifies the account relating to the individual.
6. If no account is to be found, this use case terminates. (Others may be initiated.)
7. If a user account is found, the Librarian attempts to locate the requested volumes within the library system.
8. Any books subject to recall notices involving other users are not loaned.
9. Volumes not identified within the system are not loaned. (They are physically retained by the Librarian.)
10. The Librarian attempts to record the valid loans.
11. If the date is one on which loans are withheld, the use case terminates.
12. If the user account records that the user already has too many books on loan, the use case terminates.
13. If the user account involves outstanding fines, the use case terminates.
14. If the user account records overdue books, the use case terminates.
15. Otherwise it succeeds.
16. If some of the loaned volumes are subject to recall notices in favour of this user, the Librarian enacts the Cancel Recall use case.
17. Success state changes include (i) updating the user account with the titles and volume information for the loaned books and (ii) updating the library catalogue with the information that the proffered volumes are on loan to the identified user.
18. Point 17 represents the **post-conditions** of the main success scenario.
19. Notes:
    a. Books are distinguished from volumes in that a given book may be present as several distinct volumes (multiple copies) in the library.
    b. One identifies books by ISBN number, but volumes are identified using an internal library code described in …….
    c. There are categories of users in the library which have correspondingly different borrowing allowances. Further information on this can be found in ……..

This use case description reveals that this analysis implies that other use cases are involved in its execution. We could identify:

- Log in
- Locate Account
- Locate Volumes
- Log out

There may be more[90].

[90] I've just realised I put no footnote on this page - my apologies!

© OUDCE, January, 2013

Here is a tabular version of the same use case description.

use case number: 1	**Issue Loan**
Goal	Handle the functionality associated with issuing books.
Description	Library user approaches Librarian with a verbal request to borrow books. Librarian uses the system to locate the account of the user and update it with a list of the volumes borrowed. No update is made if the account does not exist, if the user has too many books on loan already, if the user owes money from fines, or if the library is not currently issuing books.
Actors	Librarian – a privileged user who may: • update user loan information • create and delete user accounts • issue recall notices on books • update the record of library volumes Librarian must log in to the system to use it. Librarian is a privileged user and has a login password.
Constraints	This use case must execute in under two minutes with a mean execution time of one minute or less. Librarian should be able to learn the associated activities in under one hour. Screen dialogs should be readable to people with averagely poor eyesight. Screen information should be printable and accessible to members of the public.
Pre-conditions	For the main success scenario these are: • an existing and valid user account showing no money owed and no overdue books • an user allocation that is not exhausted • volumes that are not subject to recall by other users and which can be identified within the library catalogue. • that the library is issuing books
Main success scenario	The librarian does successfully issue the books by updating the user account with information on the volumes loaned. The scenario has these stages. 1. Librarian logs into system. 2. Librarian locates user account. 3. Librarian locates volumes to be borrowed. 4. Librarian updates user account. 5. Librarian updates volumes database. 6. Librarian logs out of system.
Post-conditions for main success scenario	• The system records access by Librarian. • The volumes database is updated with loan information. • The user account is updated with loan information.
Other scenarios(named in brackets)	1. The user account is found not to be registered with the library. (No user account.) 2. The user account has insufficient allocation for the books required. (Insufficient allocation.) 3. Some of the volumes to be borrowed are subject to recall notices involving other users. (Outstanding reservations.) 4. The user account is found to reflect money owed to the library or to have outstanding overdue books. (Account debits.) 5. The library is not currently lending books. (Recall period.)
Related use cases	Use case 2 – Cancel Loan Use case 3 – Create Account Use case 4 – Cancel recall Use case 5 – Log in Use case 6 – Log out All of these use cases involve Librarian.
Frequency of occurrence in the system	The main success scenario is one of the two commonest scenarios in the system (the other being Cancel Loan). It represents about 46 per cent of the activities involving Librarian.
Test generation	• Each of the six scenarios described must be tested. • The timing constraint is fairly lax but should be checked. In practice it is unlikely to be an issue in this system but for contractual reasons it may be appropriate to include auxiliary code to collect data on how long issue clerks are taking to record loans, when the alpha system is available, and this may have a bearing on interface design.
Notes	We assume that Librarian has no trouble getting into the system but any password-protected system will occasionally cause problems. We should check on library policy with respect to login problems. The following terms are defined in the system glossary • Librarian • Account • Book • Volume • User

© OUDCE, January, 2013

We have now given you two preliminary use case descriptions for the same use case. You can see that they are both incomplete. You might use one or the other in your documentation but you would certainly not use both[91]. You should also realise that both represent 'work in progress'. One would expect to modify them as our understanding of the system increases[92].

The number given in the tabular version suggests an organisational hierarchy of use cases which might be identified elsewhere. A well-organised project would take steps to ensure that these important numbers were unique and identifiable. Perhaps individual team members would 'own' individual use cases (and this might be indicated in the table above).

The tabular version mentions a 'system glossary' which clears up such things as the distinction between books and volumes (which is likely to bedevil this project). You will be able to imagine what such a system glossary would look like. It's a high-level **data dictionary** to define system terms.

One criticism of both of these descriptions is that they do seem to blur the distinction between books and volumes. It may be that project managers would insist on clarity for these terms, wherever they occur.

Both of these descriptions are fairly complicated. You would probably not wish to make the use cases they describe much more complicated, because this would reduce their communications aspect. If considerably more detail is required you should probably consider splitting the use case into simpler sub-use cases, which were described in detail.

Both descriptions might be supplemented by **activity diagrams**. Here is one that represents part of one scenario. This is the second kind of UML diagram that we show you but, unfortunately, current versions of *Violet* do not support it. Activity diagrams can be used to represent workflows in a variety of modelling situations.

[91] They could contradict each other.

[92] If you have the time or luxury to play trendy games, you might divide your team and get each half to produce one of the two accounts of the use case, then compare them and make the appropriate amendments to one of the two accounts.

© OUDCE, January, 2013

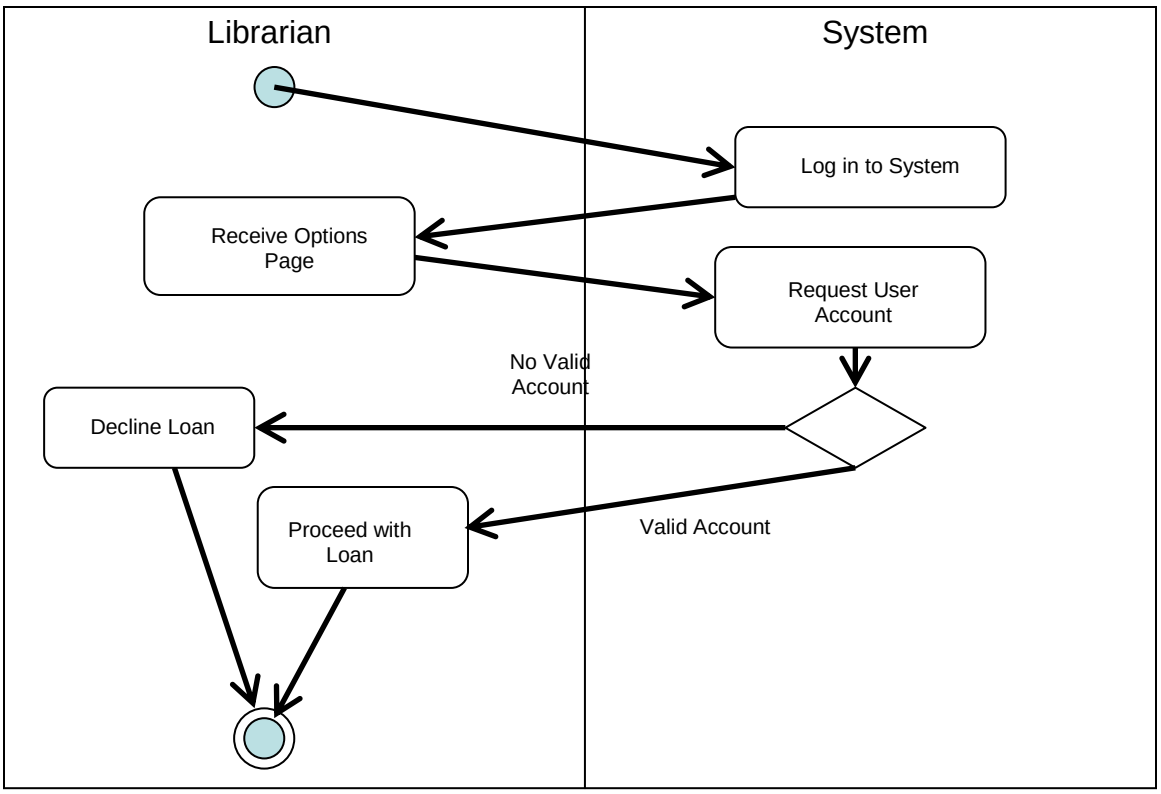


Figure 4.4: An Activity Diagram Representing a Scenario

If you used this diagram you would probably expand the 'Proceed with Loan' activity, which would encompass checking the status of the user account, obtaining database entries for the volumes, updating the user account and updating the database entries.[93]

The diagram is divided into two 'swim lanes' representing areas of responsibility.[94] There is a start state and an end state. There are a number of activities and these are linked by arrows. There is also one diamond-shaped decision point.

We hope you can understand this activity diagram even though we have not defined the symbols.

Activity diagrams do have other symbols. This one was produced first in *PowerPoint* and then transferred to a *Word* document. We shall return to activity diagrams a bit later. They are more normally used to represent workflows within the system. (The above one crosses the system boundary.)

[93] If you have already studied databases you might worry about loan information being stored in two separate places (user accounts and book databases). However, it is inconvenient to consider data design issues at this point. We shall say a little more about databases, later.
[94] Unusually, this activity diagram swim lane actually represents the system boundary.

© OUDCE, January, 2013

End note

This chapter has introduced you to use case diagrams and described how use cases are used. Nothing we have said so far restricts these ideas to object-oriented systems.

In future chapters it will be understood that our interest is in object-oriented software, but this does not apply here and you may find use cases to be helpful in many other situations.

We want to end by stressing something that we have raised before. Use case diagrams are brutally simple but determining use cases is not. Use case diagrams are simple artefacts that are the result of intense and complicated analysis. Don't let their simplicity fool you into thinking they are trivial or that they can be produced without a lot of hard work.

Some of that hard work will show up when you produce your use case description. One can gauge the work done in terms of how vague, complicated, contradictory or incomplete that document is. Get into the habit of subjecting your own use case descriptions to violent critical assault and expect to take a lot of time in producing them.

In the chapters that follow, we imagine that a complete set of use case diagrams for the Library system has been constructed.

© OUDCE, January, 2013

Chapter 5: Class Diagrams

We hope that you now understand how use cases can be used to identify what sort of things use system services,[95] and what sort of services systems are to provide to users.

Use cases are wish lists. They say what we want to happen, not how it happens. In this chapter we shall move closer to the *how* of things by considering the objects that the system should contain and their relationships.

Once you have an acceptable set of use cases to characterise the system services, you can use them to model the system further and to begin to develop other aspects of its design.

This analysis and design process is bound to impinge on your use cases. You should expect to have to make changes to them as a result. Analysis is always an iterative process.

You will need to have a clear understanding of the basic ideas of object-oriented systems in order to make sense of the following chapters. You should start the work on this chapter by reviewing the summary at the end of Chapter 2 and making sure that you are familiar with what it says.

Association, responsibility and inheritance

These points need to be covered when building an object-oriented system.

5.1 The objects that occur are identified, together with their attributes, interfaces and methods.

5.2 Families of related objects are classified into class families based on commonalities of attribute and behaviour.

5.3 Specific system functionality is vested in the behaviour of objects, as services are translated into **collaborative** sequences of messages between objects and the specific **responsibilities** of individual objects.

5.4 The objects that need to know about one another in order to communicate with each other and discharge their responsibilities are identified.

5.5 This knowledge is translated into **associations** between classes, which are usually expressed by defining instance variables typed to the classes of the objects with which communication must be achieved.

Object-oriented analysis and design methodologies offer techniques that help you with each of these points. The above points are not to be thought of as being performed in strict sequence because they are heavily interrelated.

Point 5.1 is already familiar to you from our discussions in Chapter 2. One can normally get started with it by studying the system specification, because the nouns used will often correspond to valid classes.

In the case of the Porterhouse Library, such a study might suggest the following related classes.

[95] Remember that actors do not have to be human and might well include other systems.

© OUDCE, January, 2013

Candidate classes for the Porterhouse Library system.

5.6 Book
5.7 Copy
5.8 Catalogue
5.9 Librarian
5.10 Head Librarian
5.11 Catalogue
5.12 Student
5.13 Staff (distinct from Librarian, that is the 'Senior' members of Porterhouse,[96] its academic staff)
5.14 Users (describing the attributes and behaviour of Senior and Junior college members)
5.15 Loan

Not all will be appropriate and there are clear relationships between these classes. Student is a kind of User, as is Staff, which suggests inheritance. Book and Copy might be related by inheritance except that the library probably does not have such a thing as a Book[97] – it consists entirely of *Copies* of books. The point is that a single book might exist as many distinct copies and that much of the processing will involve these individual copies. One borrows a copy of a book, but one reserves a book (and will accept any copy of it).

We thought about this and decided that Copy was a poor name for the class we had in mind, so we replaced that with the term Volume. If this class is actually used by us in our system, the notes associated with it will record that it represents what the specification refers to as Copy and that Catalogue must be an aggregation of Volume(s). Some books will normally exist, however, as collections of sub-books which are *commonly* called volumes. (We are thinking of such things as encyclopaedias and dictionaries.) Consequently, our use of the term 'Volume' might also need to be further reviewed[98].

Point 5.2 begins to emerge from this sort of preliminary analysis.

Point 5.3 can be addressed by reviewing the existing user cases. For example, consider the work we started on the Borrow Book use case, in Chapter 4. Our final account presented you with this sequence of events.

Steps in the Borrow Books use case (Use case 1

Step 1.1	Librarian logs into system.
Step 1.2	Librarian locates user account.
Step 1.3	Librarian locates volumes to be borrowed.
Step 1.4	Librarian updates user account.
Step 1.5	Librarian updates volumes database.
Step 1.6	Librarian logs out of system.

Some of these steps also involve further use cases. For example, use case 5 (Log in) is associated with Step 1.1, while Step 1.2 may involve a use case called Locate Account. We presented a UML use case diagram showing that in Figure 4.3, but we don't seem to have included it as a related use case in our tabular representation for Issue Loan (use case 1). This mistake is deliberate: in real life we would have to go back to the table and decide whether or not

[96] The requirements definition document speaks of 'senior staff' so we use the term here.
[97] Another way of putting that is to say that there are no objects from the Book class: it is an *abstract* class.
[98] And we have to say, on the final review of the first version of this unit (November, 2005) we came to believe that Volume was probably the wrong name - but one has to settle on something, and that is what we had settled on, in the Summer.

© OUDCE, January, 2013

to make an adjustment. If we decide not to have that activity as a use case, the diagram presented in Figure 4.3 should be removed.

Walkthrough is a term that can acquire a definite meaning in some object-oriented methodologies. We use it informally to mean the sequential consideration of a series of steps.

A walkthrough of the above sequence will present you with a preliminary understanding of how the candidate classes identified in Point 5.1 might be expected to interrelate. It will also suggest other classes that the system will need.

For example, in Step 1.1, Librarian logs into the system; this is going to involve objects handling user names and passwords and some sort of dialog with an object that controls system access, perhaps by arranging the display of particular dialog frames. None of our candidate classes fits this controlling role, so we are going to need to add to our list of possibilities.

This is the first inkling we have that we need another class.

It may be useful to do walkthroughs on the main use cases, especially when you work in teams. One might analyse this use case by doing a walkthrough, and noting how each step imposes *responsibilities* on the classes it is likely to involve.

Walkthroughs may be useful when considering other UML diagrams such as activity diagrams or sequence diagrams.[99]

If you use walkthroughs, do take them seriously. Allocate roles to the different members of your team, and use one team member to record what happens, and any other considerations that occur[100].

When we get to Steps 1.4 and 1.5, which are:

Step 1.4	Librarian updates user account.
Step 1.5	Librarian updates volumes database.

we might worry about whether these might usefully be treated as further use cases. At present, we regard them as part of the functionality of *this* use case. Perhaps that will change.

Whether it changes will depend on how far it seems reasonable to regard either of these things as separate actions in their own right. Our *analysis* attempts to break things down into the smallest possible components. If other use cases have these as constituents, it might be better to consider them separately.

Certainly, these steps are going to involve messages being sent – perhaps to user account objects. Librarian is not a system class or object (since it is an actor), but which object sends these important messages?

This is the second inkling we have that we need another class.

[99] Which we have not yet covered!

[100] Walkthroughs sometimes dissolve into giggles, watch carefully – if points are repeatedly glossed over, something is going wrong.

© OUDCE, January, 2013

Responsibilities

Point 5.3 introduced the term **responsibility**. Responsibilities relate to things that you rely on classes (or rather, their objects) to remember or do, in order to achieve system objectives.

One popular object-oriented analysis and design methodology seeks to analyse systems in terms of the responsibilities vested in their constituent objects. Authors offer a number of classifications of the various kinds of responsibility that can occur, but they basically come down to storing or providing information, that is:

1. Responsibilities for 'knowing' about some other system objects (that is, being in a position to send messages to them).
2. Responsibilities for providing some sort of service (that is, returning some sort of useful information or doing something).

As we shall explain, the first responsibility might involve an **association** between classes; the second responsibility might well involve some sort of coded calculation and **collaboration** with other objects of the system.

Step 1.1 involves an actor called Librarian logging in to the system. This will probably necessitate a dialog controlled via a login screen. The information proffered should grant access to the system.

At this point we might wonder whether the system has many kinds of user, each with an individual password and login name. Alternatively, there may be one simple password that permits anyone who knows it to act as 'Librarian'. The specification speaks of Librarians, and the *Head Librarian* and there is a clear suggestion that these system users are to have different powers.

One cannot tell which situation obtains from the specification we have. It simply speaks of 'access control via user name and password'. The ambiguity can only be resolved during the course of discussion with the system commissioners, and they are likely to want as much for their money as they can get. Commercially, your job would be rather different from that – to deliver software that does what you agreed it would do and not very much more!

Another factor, when building systems, is to start by offering basic services. Perhaps we should leave different sorts of user account to later versions. Of course, we ought to try to make it reasonably easy to amend the system to cater for this later. So, in that sense, we shall inevitably have to make some provision for it in our original design.

However this is resolved, user name and password information must be stored somewhere in the system. Here are some possibilities.

- There is exactly one user name and one password and these are stored as attributes of a class or object.
- There are many user names and passwords, perhaps relating to different capabilities within the system. They are all stored as attributes of a class or single object.
- There are many user names and passwords, perhaps relating to different capabilities within the system. They are stored as attributes of different objects from the same class.

The questions are:

- Which object(s) have the responsibility for storing user name and password information?
- Which object(s) have responsibility for handling login dialogs?

© OUDCE, January, 2013

Further, *how* should these objects collaborate? Should we have one object for each user account, or a single object that handles all the user accounts?

- One solution that occurs to us is to have a class called Librarian.
- This class would have attributes that identify a user name and a password.
- Each object from the class would be another 'Librarian' – with some defined set of capabilities relating to the system.
- Login would involve checking each of the Librarian objects in turn, to see if the proffered user name and password was matched by the corresponding attributes of any one of them.
- In the event that it was, the object in question would arrange for the next dialog screen to appear; in the event it was not, the object doing the checking would present an error screen.
- These different Librarians could each have different capabilities (in relation to the system).

This is beginning to sound like an attractive solution. We could start our design by giving them each the same capabilities and then build in changes as our system becomes more complicated.

But we don't really seem to be talking about Librarians here. Some of the instantiated objects from the class might correspond to the specification description of a librarian, but others might not. And how are we to implement this process of 'checking each of the Librarian objects'?

Moreover, do we really want a whole host of objects to be displaying screens on the basis of logins? Won't this embed the user interface processing firmly into the code of the system in a way that is likely to result in a maintenance nightmare?[101]

This is the final indication that we need another, dialog-organising class.

- It seems more appropriate to have some kind of administrator class that has the responsibility for 'knowing' about the various kinds of system user.
- The administrator would send the password information to each system user in turn, and if one of them replied that the password was correct, the administrator would organise the display of an appropriate screen.

Our system probably does require there to be a number of categories of user, of which Librarian is just one. One could control the capabilities of these categories of user by arranging the display of different screens with different dialog fields, depending on what the system user is permitted to do. Our administrator will handle that, and the code relating to user interfacing will be firmly in one place, making maintenance much easier.

[101] Modern thinking tends to separate interface questions from processing ones. We shall say a bit more about that when we discuss the MVC paradigm, later.

© OUDCE, January, 2013

Summary

Our analysis so far has led us to conclude the following things.

1. There should be a class called something like Admin. Objects from this class would control system access. They have the responsibility of 'knowing' about all the permitted users of the system.
2. There should probably be only one object[102] from the Admin class in the system. This object will be involved in some dialogs with system actors. In particular, this object will handle the login dialog.
3. The system will need to have various categories of user.
4. User objects will have the responsibility of 'knowing' password and login details.
5. A given actor logs into the system by supplying the Admin object with a user name and password.
6. The Admin object performs password checks by sending a message to each User object in turn, offering the proffered user name and password.
7. User objects will have the responsibility to reply to these messages, either agreeing that the user name and password is correct or indicating that it is incorrect.
8. So, User objects will *collaborate* with the Admin object during login.
9. The Admin class object denies access to the system if every User object replies that the user name and password are incorrect.
10. The Admin class object admits the user to the system if any one object responds that the login details are correct. This setup could also be used to permit any one user to be logged in to the system no more than once at the same time.
11. When a login is permitted, the Admin class object has the *responsibility* to display an appropriate screen to the user offering the facilities that his or her user details permit.
12. In the case of Librarian class objects, these activities include interrogating the books database, updating user accounts with loan and return details, and so on.

Although this is nothing to do with the use case we are analysing, we might note that our Admin object could control the creation of user accounts. Deciding whether this is appropriate or not awaits an analysis of other use cases, associated with system maintenance and privileged users. We would include something about this in our system diary, because maybe these ideas are addressed elsewhere in our team, and we should revisit the idea later.

We can express what we now have using this UML class diagram.

[102] OO languages provide simple facilities for limiting the number of objects instantiated from a class. A good place to do that would be in the constructor.

© OUDCE, January, 2013

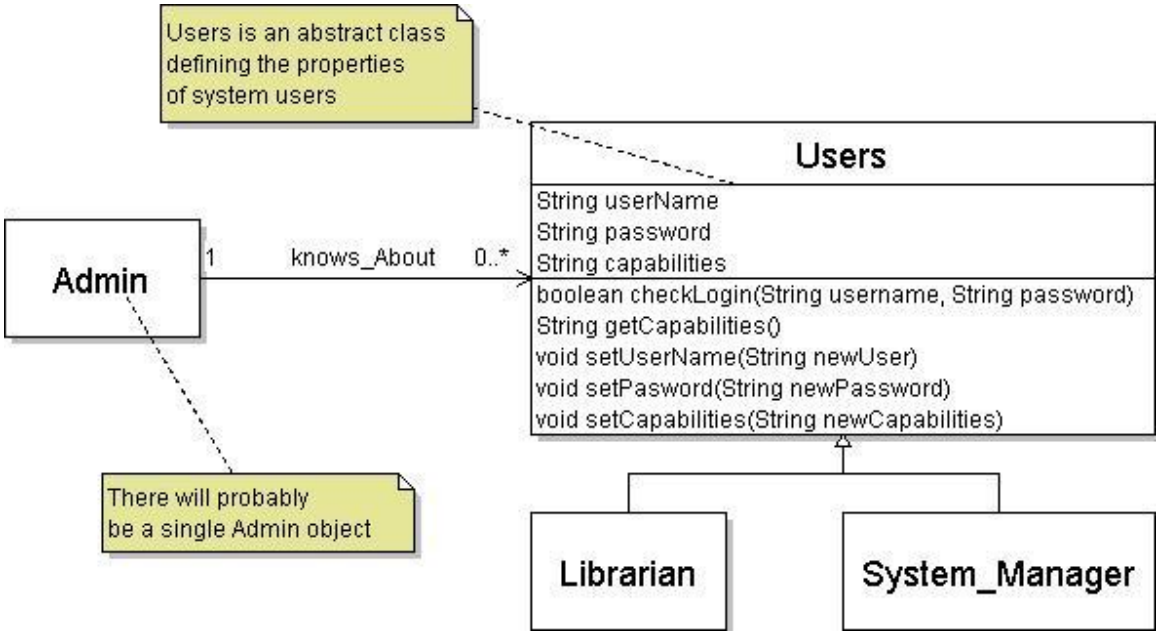


Figure 5.1: Class Diagram Obtained by Analysing the Login Use Case

This class diagram includes four classes and shows that the Admin class 'knows' about users of the system. We expect there to be several different kinds of user of the system. Users class describes features one would expect any user to have. Librarian would be a-kind-of Users. Immediately, we regret that plural! Perhaps User would be a better name! Right, we shall change it!

Members of the class **User** store user names and passwords and have methods that allow other objects to organise attempts to log into the system. We have actually described three attributes of User objects, in this diagram, and four pieces of behaviour.[103]

We have followed Java function signatures to do this, so please recall that the method called checkLogin will expect to receive two String objects: one identifying the inputted user name and the other identifying the inputted password. The method will reply **true** if the login details fit this user (that is, they match the values stored in user name and in password) and **false** if they do not.[104]

UML is not beholden to Java and actually expresses this information slightly differently. We shall introduce the notation UML favours in Figure 5.3.

The class User is a superclass of two other classes. The first is called Librarian and is supposed to cover the properties and abilities associated with the librarians described in the system description. The second is called System Manager. The need for this class has been deduced from the system specification because clearly a privileged user will be needed to set up user names and passwords.

[103] There is no need to list all possible attributes and behaviours in class diagrams. Remember that these techniques are supposed to communicate rather than impress!

[104] In point of fact the only information one might conceivably get from Figure 5.1 is the types of parameter that have to be sent to checkLogin and the type of parameter that returns. The meaning of what is going on actually has to be described elsewhere.

© OUDCE, January, 2013

We may well need to have various categories of privileged user since we shall need to do such things as:

- set up and remove user accounts
- set up and remove accounts for privileged users such as Librarians
- modify the book and volume data
- maintain the system

We are considering implementing Head Librarian as a subclass of Librarian (so it has all the attributes and behaviour of Librarian, plus additional powers of its own).

Discussion of these categories of privileged user must be postponed for the moment, but one might note the above points in our system diary.

You can see that the User class has methods that set up passwords and user names. Such activities are clearly privileged and one would expect some security checking in the code associated with them. We don't have time for that here.

Here are some points from the analysis so far.

1. The User class does not itself correspond to any user of the system. It merely defines properties and behaviour that are to be common to *all* users.
2. There will never be objects instantiated from the class User. In object-oriented programming, classes that do not have objects instantiated from them are called **abstract classes**.
3. We have noted that Users is abstract in our UML diagram.[105]
4. Two other classes, Librarian and System Manager, do inherit from Users. These classes may have instantiated objects.
5. In object-oriented programming, classes that can have objects instantiated from them are sometimes called **concrete classes**.
6. The system specification also talks about 'head librarians' as having special abilities. We can expect there to be a Head Librarian class. Perhaps it should be a kind of Librarian?
7. Figure 5.1 says that Librarian and System Manager both inherit directly from User.
8. So Librarian is a kind of User.
9. We may be confusing you with this discussion of a Librarian class. We already had an actor called Librarian and we made it very clear that that actor lies *outside* the system boundary. Now we have a class called Librarian that is clearly *inside* the system boundary. What is going on?
10. We could answer that the class called Librarian describes the system abilities that the actor called Librarian gets by logging in to the system, but this is dodging the issue. We should either rename the actor or rename the class.
11. We choose to rename the class and incorporate these points in the following UML class diagram.

[105] UML provides a better way to indicate abstract classes – by including <<abstract>> in the box giving the class name. This seems to be quite hard to achieve using *Violet*, so we have done it by means of a note.

© OUDCE, January, 2013

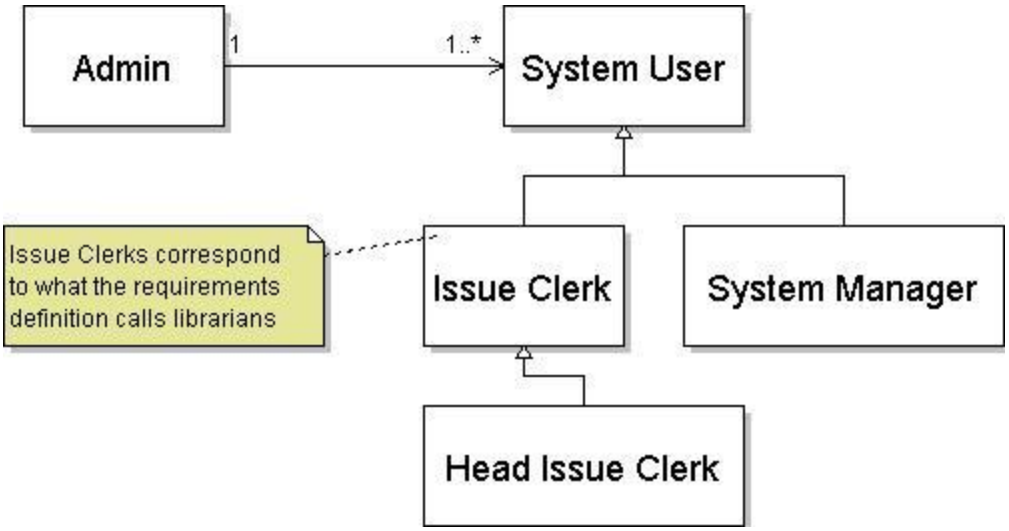


Figure 5.2: Revision of the 5.1 Class Diagram

We have left out the attributes and behaviour that we noted in the previous class diagram.

Figures 5.1 and 5.2 are examples of UML **class diagrams**.

Syntax notes on class diagrams

1. Classes are represented as boxes containing class names.
2. These boxes may additionally be subdivided to show attributes and methods.
3. It is generally useful to show only those attributes and methods that are directly relevant to your use of the class diagram.
4. The parameter type information we showed you in Figure 5.1 would probably be represented rather differently in standard UML (see Figures 5.3 and 5.4).
5. Class diagrams can involve several sorts of arrow, representing such things as association or inheritance.
6. Arrows are distinguished by their heads.
7. The multiplicities and names of associations may be shown on class diagrams.

In the main, one would use a class diagram to represent some specific feature of a class and suppress information not directly relevant to that feature. All UML diagrams should be supplemented by additional textual information. For instance, one might supply more complete information about a class by means of a CARC card (see below).

Class diagrams show the classes of the system and how they relate to one another. You have already seen them in Figure 2.9. Figure 5.2 illustrates how UML depicts inheritance. Head Issue Clerk is a kind of Issue Clerk, and Issue Clerk is a kind of System User.

In a real-life situation, one would at this point check that librarians don't object to being described as issue clerks![106] If we want our system to be acceptable, we should take pains not to offend the people who are going to be using it.

[106] The foot soldiers can kill a delivered system by refusing to use it. There is often a distinction between those who commission a system and those who use it. Both sides need to be happy. This is something that you simply have to take into account – unless you work for a huge arrogant multinational!

© OUDCE, January, 2013

Figure 5.2 also shows another kind of relationship between classes – the one between Admin and System User. This relationship is known as an **association**. Associations are relationships between classes that relate to system needs.

The presence of an association normally indicates that objects from one or both of the classes can communicate with objects from the other class and that in turn often translates into classes having attributes that will be references to objects from the associated class.

This particular association reports that one Admin class object knows about one or many System User objects.

We sneaked the same notation into Figure 5.1 but there, for some reason, the association involved between zero and many objects. This sort of inconsistency is common in the early stages of analysis. One would hope the team would pick up on it and rule one way or the other. In this case, we supposed that it was decided that one had to cater for situations in which there were as yet no users in the system so **0..*** was preferred over **1..*.**

One might then ask how one enters a user into the system, because our present view suggests this is a privileged action by a system manager user, and if there is no system manager this is impossible! We shall leave this argument unresolved for now, but still settle on 0..*.[107]

We actually skirted round associations in Chapter 2, when we mentioned that in order to send messages to each other, objects needed to 'know' about other objects. Now we shall consider associations in more detail. In doing this, we shall cover the remaining points, 5.4 and 5.5, that we identified as needing attention when building a system.

Admin class is our main entrée to the system. We can imagine that one activates the Admin object by entering login details into a screen.

The Admin object acquires this data (as we shall explain in Chapter 6) and begins a process of sending messages to the System User objects.

Admin has the responsibility for 'knowing' these objects, that is, there is an association between Admin and System User called **knows_About**. Here is a UML diagram showing that.

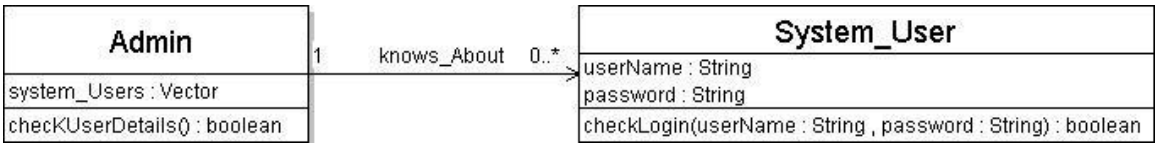


Figure 5.3: The 'knows_About' Association, Linking Admin to System_User

You can see the 'knows_About' association in this diagram. It involves one Admin object and anything between zero and an unspecified high number of System_User objects.

We have changed the name of the System User class by connecting the two words with an underscore character because most object-oriented languages do not permit single identifiers (names) to include space characters.

[107] And the present author records uneasiness in his system diary!

Properties of associations

- They are usually **navigable**. (This one takes us from Admin to System_User, but some associations go both ways.)
- They have names. (This one is called 'knows_About'.)
- They have multiplicities.

You have probably met the idea of multiplicities before. It occurs in database theory and in entity-relationship modelling. The idea is to express information about the numbers involved in associations. In this case, the multiplicity is one to many, with one to zero being possible.

UML permits you to express multiplicities using the notation shown above.[108] One expresses each side of the relationship by the small numbers shown. These numbers can express exact figures, like the left-hand side of knows_About, or they can express an exact or an indefinite range.

In the system we are building, navigability should translate into a route to any class in the system from the Admin class. A complete class and association diagram ought to show that. If one of the system classes does not lie on such a route, then it is impossible to send messages to it and there may be something wrong with our system.[109]

Figure 5.3 also gives some of the attributes and behaviour relevant to the knows_About association. We have now adopted formal UML notation so you should contrast this diagram with Figure 5.1.

UML is unrelated to Java and does not use Java's notation for representing the class of objects returned by methods or the class of attributes. We imagine that you will find its conventions it does use self-explanatory, but note the colon and the fact that type information occurs on the *right* rather than the *left* -

In place of Java's	**String myString,**
UML would have	**myString : String**

Although one would need additional documentation to specify it precisely, Figure 5.3 indicates that the Admin class object will receive the message checkUserDetails() in response to the submission of a user name and password. The method will have access to both these proffered pieces of information (as outlined in Chapter 6) and it works by sending the message checkLogin(username , password) to each of the System_User objects in turn.

It knows about these objects because it has a record of them in the system_Users Vector. (Vectors are discussed in Chapter 2 above.)

One could imagine the code looping through each of these objects, first retrieving them from the Vector, and then sending the message. Each message generates a boolean reply. Login succeeds only when one of the objects replies with the boolean value **true**. Of course, such a reply terminates the round-robin process of message-sending.

[108] People who have previously studied SSADM may be aware of other notations, for example, 'crows' feet'.

[109] This observation is broadly correct but it is also possible to connect objects to classes associated with the system interface and isolating some classes is occasionally a good idea for security reasons. This will crop up when we discuss how the Admin object obtains information from the password screen.

© OUDCE, January, 2013

We can represent this searching process using the following UML **sequence diagram**, which shows two representative password-checking messages being sent by the Admin class object. The first is shown as failing. The second is shown as succeeding.

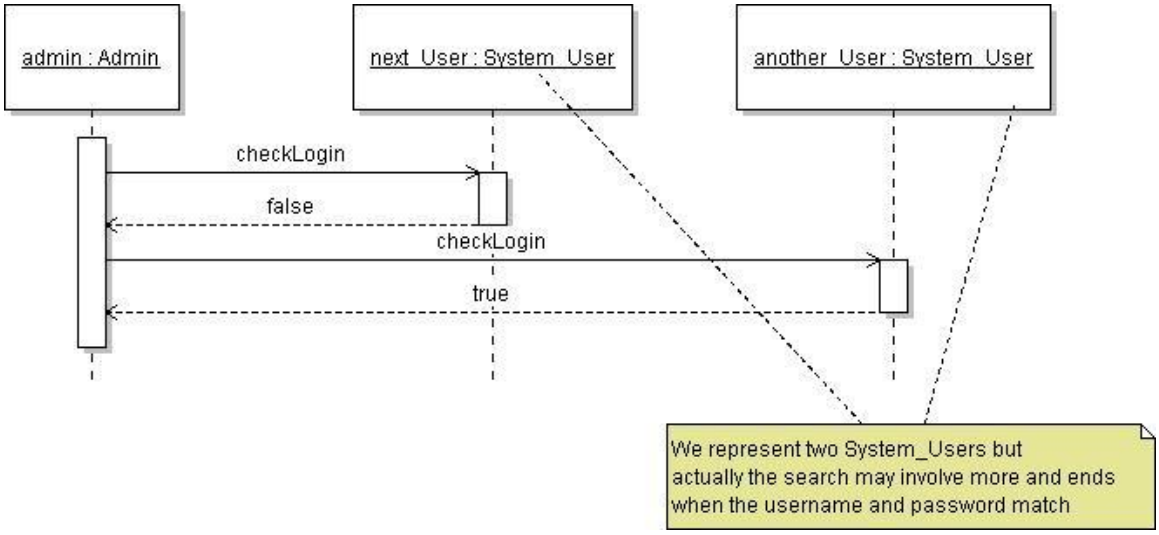


Figure 5.4: Sequence Diagram showing checkLogin Message

UML sequence diagrams show objects and messages. You will recognise that next_User : System_User is how UML identifies an object referenced by next_User and coming from the System_User class, but since we mentioned earlier that the class itself was abstract, this might trouble you.

In point of fact, the next_User object is a-kind-of System_User, and would come from one of its subclasses. Figure 5.3 identifies three such subclasses. It is entirely valid to describe next_User as being in the System_User class, because it is a-kind-of System_User.

We are not claiming that next_User is actually an object from System_User - it is an object from a descendant class.

The sequence diagram we have shown depicts objects, messages and replies. The replies are shown as dotted arrows. In practice, UML sequence diagrams often do not show replies and they normally involve a series of messages sent to a series of objects. We shall show you more sequence diagrams a bit later and indicate other uses for them.

This diagram also introduces you to the way that we shall refer to the single object in the Admin class in this unit: **admin : Admin**. This is the UML notation for indicating an object called admin that belongs to the Admin class. You will see this a lot in subsequent chapters and should note it.

© OUDCE, January, 2013

CARC cards

Responsibility Driven Design is an object-oriented design methodology that begins with an analysis of what a system does in terms of the allocation of responsibilities to individual classes. Classes discharge responsibilities by collaborating with other classes. One achieves a high-level account of what happens in terms of these sequences of responsibilities and collaborations. Of course, one must then design software structures to implement these decisions.

One of the ideas in this methodology is to develop **CRC** cards for each class – **Class, Responsibility** and **Collaboration** cards. Here is a CRC card for Admin class, as our analysis currently understands it.

<table>
<tr><td colspan="4">Admin <<concrete>></td></tr>
<tr><td colspan="2">Date</td><td colspan="2">8 April 2005</td></tr>
<tr><td colspan="2">Version of card</td><td colspan="2">1</td></tr>
<tr><td colspan="2">Author</td><td colspan="2">Bob Lockhart</td></tr>
<tr><td colspan="2">Superclasses</td><td colspan="2">None</td></tr>
<tr><td colspan="2">Subclasses</td><td colspan="2">None</td></tr>
<tr><td colspan="2">Description</td><td colspan="2">The Admin object is intended as handling all system dialogues. It controls user access on the basis of passwords and pre-set capabilities by displaying appropriate dialog frames offering particular facilities.</td></tr>
<tr><td colspan="2">Responsibility</td><td>Collaborator</td><td>Association</td></tr>
<tr><td colspan="2">1. Knows all the objects in System_User class
2. Checks proffered user name and password with members of System_User class</td><td>System_User</td><td>knows_About

a 1 : * association</td></tr>
<tr><td rowspan="2">Interface Methods</td><td>checkUserDetails() : boolean</td><td colspan="2">Called by the system interface in response to user name and password submission</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td colspan="4"><u>Notes</u>
There will be a single object from this class in the system. It will connect to the user interface and be the channel for user communication with the system.</td></tr>
</table>

Our CRC cards are non-standard in that we have included a great deal of additional information not usually associated with CRC cards. One important piece of additional information we have

included is class associations. One might speak of **CARC** cards – Class, Association, Responsibility and Collaboration cards.

We have also included a list of interface methods, that is, the methods used to communicate with objects from the class. We only know of one of these methods so far, but we would expect to add all the others as our analysis progresses.

Clearly, the information on these cards will change as the system develops. You should expect to have a sequence of the cards for each class, representing the stages in your analysis of the system, and that is why we have included date and author information with them. You keep old versions of these cards because they help you to track decisions.[110] The version numbers on each card will allow you to check that you have all of them and your system diary should record when the various versions of each card were produced.

It is generally important to support UML diagrams with textual information. CARC cards go some way towards doing that but you must be prepared to supplement this information about classes with whatever additional documentation may be required.

We have been speaking of 'cards' and have in mind paper documents stored in a central place and forming an important part of the documentation associated with the system. These cards might even constitute a deliverable product because they would be very useful in maintenance and revision exercises.

Of course, you might want to have electronic cards of some sort. The cards must then be accessible to the system designers, clearly expressed, and meticulously completed. One has to know *who* wrote *which* card *when.*[111]

System glossaries

It can often be helpful to develop a system glossary as your analysis proceeds.[112] One might start by including all of the terms used in the system specification, along with explanations of what they mean (which would be subject to later revisions). You would go on to record the classes you decide on, how they are used, their attributes and their behaviour.

In so far as it is at all possible, your glossary should use precise, natural language rather than design notations. Don't simplify to the point of barbarity but do try not to use jargon or acronyms in your explanations[113].

With the glossary in place, a developer analysing a different part of your system would be able to check that an identifier he or she wishes to use is not currently being used for a quite different purpose. This adds to the clarity and simplicity of the analysis and design process.

[110] One has to experience the joy of arguing all day to preserve a facility that has in fact already been removed from the system to understand the importance of decision tracking!

[111] As a development progresses people become more lax about maintaining system documentation. You can tell a good manager by looking at this documentation and whether it is maintained.

[112] Some people would refer to this as a 'data dictionary', but we resist the term because it tends to have a more formal meaning in some design methodologies.

[113] People simply won't cross-reference unknowns – they are too busy. The result is that they think they know what you have said but actually have it completely the wrong way round.

© OUDCE, January, 2013

There is no harm in referring to specific products of the development process in the system glossary. One might, for example, give a reference to the CARC card written by Podge Brassica on 8 April which describes the Admin class, but references of this sort should support the glossary rather than comprise it: the glossary itself would be in natural language; and there is no point to such references unless that which is referenced is immediately accessible.

You should now be in a position to construct a CARC card for System_User and you might like to sketch one out now.

Products of the analysis and design process

Although the functionality behaviour of the system was an important consideration in this chapter, our end-product – class diagrams – relates to a static system structure. The dynamic aspects of the system are modelled by other diagrams, such as sequence diagrams. We give a taste of these here and return to them in later chapters.

When you use these modelling techniques, do it conscientiously. You will be tempted to imagine that you have a clear understanding of particular classes and therefore do not need to draw class diagrams or fill in CARC cards.

Our experience is that a modelling technique should be used consistently and that subconscious understanding is very often confused. If your project is using CARC cards, **do one for every class in the system**.

Our advice is to decide in advance on the techniques that one will use and how one will organise documentation. You will certainly be using class diagrams and you are likely to draw a considerable number, often involving the same classes. You need to be able to refer to them easily and have some system for tracking versions. It often happens that team members get out of step because they are using different versions of the same class diagram. Make sure this does not happen to your team!

> You need to attend carefully to the management of the development process. Management is not at all easy and that is why good managers are (sometimes) well paid.

Chapter summary

This chapter presented the key information required to model object-oriented systems. You should now have the following:

- an understanding of how the system requirements specification can get you started on deciding the classes you need and their possible inheritance patterns;
- an understanding of how to conduct walkthroughs of use cases in order to get information about classes, associations, responsibilities and collaborations;
- the ability to represent the classes in your system, their associations, attributes and behaviour, by using class diagrams;
- the ability to record information on individual classes using CARC cards;
- the ability to use CARC cards to track analysis and design decisions;
- an appreciation that the analysis and design process is repetitive – that decisions are revised or reversed as knowledge of the system grows;
- an informal understanding that the use and construction of analysis and design products need to be managed;
- some understanding of the role and importance of software development management.

© OUDCE, January, 2013

Addendum

We discussed associations in terms of message-passing but we should not permit you to think that object-oriented functionality is to be seen entirely in those terms. There are programs in which an object achieves something simply by invoking the constructor of another class without needing to store the object produced, since the class constructor has sufficient processing to achieve the desired objective.

This seems to us to be a perfectly valid association between classes, but not one that relates to the direct storage of object references. The message is that associations are links between the objects of different classes, that these links are often in the form of object references (as they will be in the cases we have discussed so far), but that other sorts of link can and do occur.

Some part of the confusion relates to whether or not constructors can be deemed to be interface methods. Constructor methods represent a sort of class behaviour (the production of new objects) rather than the behaviour of individual objects of the class. They are not really **instance methods** and are consequently not usually thought of as part of the interface to objects. On the other hand they are usually public methods[114].

We use constructor methods extensively in the library system we are designing. Our **main** function (which is where Java starts processing) begins by constructing the key admin : Admin object, which, in turn, constructs system_user : System_User objects and maintains references to them. It is these references that are used in the messages shown in Figure 5.4.

[114] And, as we mentioned rather drily a little earlier, in Java, constructors are technically not regarded as methods at all.

Chapter 6: User Interface Considerations

Our case study of the Porterhouse Library has suggested the need for a class, which we have called Admin, to handle data entry.

We have only really looked at logging into the system, which is part of use case 1 and may be part of other use cases. It is likely that our understanding of Admin class will change and refine after analysis of the other use cases.

We mentioned that the admin object (and there is only one) would receive data from the login frames, and we have explained how it would process that data (Figure 5.4). We refer to that object as admin : Admin, to reinforce its role as the single object in the Admin class.

We have been vague about how the admin object obtains its data from the frames. Indeed, the scoping aspects of our use case diagrams have tended to suggest that the frames are somehow outside the system we are developing.

This is clearly not the case. What is true is that interface issues are often considered in isolation from the main processing part of a system – the **domain model**.

In this chapter we shall outline the notion of **domain models** and **separable user interfaces** and illustrate how user input data might arrive at the admin program if the system were coded in Java.

WIMPs

Object-oriented programming grew up at the same time as Windows programming. Graphical user interfaces (GUIs) are omnipresent in object-oriented systems, whether as development environments or as integral parts of delivered systems.

Most object-oriented languages offer rich sets of facilities for developing GUIs, reflecting the facts that developers need to construct them, that this would ordinarily involve an astonishing amount of complicated coding, and that users tend to expect graphical environments having basic commonalities ('look and feel').

Modern user interfaces come 'ready made'. One develops a system and then simply connects it to pre-written display facilities. This exercise can even be carried out by visual 'drag and drop' methods. Indeed, one can construct interfaces without underlying functionality in order to test out proposed designs with the intended customers. This is an important technique (known as **storyboarding**).

An inescapable consequence of what we have been saying is that the interface part of a system can often be treated as separate from its underlying structure. More than that, one could imagine a single system having a series of completely distinct interfaces – perhaps to cater for people with disabilities, or for disjoint sorts of system access (Internet or local, mobile phone, PDA or laptop).

One tends to think of interfaces in visual terms but this would not always be the case. Indeed, other sorts of interface are almost mandatory in some settings in order to cater for people with disabilities or specific needs.

This is the important modern design principle of having **separable user interfaces**, but our treatment of it stresses the subservient fact that interface design can be viewed as a pure HCI issue rather than one having ramifications for system functionality, analysis and design.

Modern Windows-based systems offer a standard set of facilities (often referred to as **WIMPs** – Windows, Icons, Menus and Pointing devices).

© OUDCE, January, 2013

These include:

- textboxes, into which users may enter textual data
- password boxes, or subclasses of the above in which user input is not directly echoed
- menus, or dropdown lists of possible facilities (sometimes leading to further menus)
- check boxes, which permit the selection of a subset of available options
- radio boxes, which permit the selection of a single choice from several
- buttons, which control operating functionality

and much more. You will recognise these notions from your experience of Windows, the Web, or perhaps if you have done any programming using Java Swing.[115]

One would construct an interface on the basis of HCI ideas, try it out on the intended users, and make such amendments as seemed desirable before finally associating it with the delivered system. The first attempts would not usually be associated with functionality; they would exist purely to gauge the reaction of those who have to use the final system.[116]

In Java, the interface would consist of a number of related Java classes and their objects. The classes are an integral part of any Java development environment and come to you quite free of charge.

The objects can generate special Java objects known as **Events**. They would do this in response to user actions.

To illustrate this:

1. Java JButton objects generate ActionEvent objects when users operate them.
2. One can register other objects as 'listeners' for these ActionEvents.
3. Each of these other objects must possess a special 'event handler' method with a predefined name.
4. In the case we are describing, that handler would have the signature:

```
public void actionPerformed(ActionEvent e)
```

5. When the JButton is operated, the system uses that handler to send the listener object a message including the ActionEvent object.
6. What the listener object does in response to this message is entirely its own business.

In our library system one could imagine a frame displaying a textbox for the user name and a password box for the password. There might be a 'submit' button. Pressing that would cause an ActionEvent object to be generated and the actionPerformed message to be sent to the Admin class object.

[115] Visual interfaces tend to reflect the look and feel of the underlying operating system. Certainly, all interface software must interact with that at some stage. Swing is an attempt to restrict interaction to the minimum possible, with much of the GUI structures coded in pure Java. In that way, it was hoped that Java GUIs would look very similar in all systems.

[116] Remember that the people commissioning the system are likely to be different from those that have to use it. You have to satisfy both of these conflicting interests to make your final product successful.

© OUDCE, January, 2013

This presupposes a number of technicalities. These are that the Admin class:

- **implements** the Java ActionListener **interface;**
- includes code for an **actionPerformed** method;
- ensures that its object (admin : Admin) registers itself with the submit button using the **addActionListener** message.

It may well be that this technical information is mysterious to you[117]. If so, don't worry, since we only included it to tell a reasonably complete story. You only need to understand the general idea of information passing from frames to the admin : Admin object, so no part of it is needed in what follows. In any case, our test case analysis is supposed to be independent of any programming language and the story would be similar, but different, in C++ or Visual Basic.

The registration of the admin : Admin object with the submit button sets up an association between the **screen widget**[118] (in this case, the button) and the single object from the Admin class. The multiplicity of this association would be 1:1.

Java interfaces

The three points above include information that may be unfamiliar even to those with previous experience of Java. They mention classes *implementing* interfaces but the interfaces in question are a little different from the interfaces we have been discussing, and the term 'interface' is used in the third of the senses we described in Chapter 2 (point 2.3).

A Java interface is a sort of contract between a program and a set of predefined facilities which effectively allow classes to 'inherit' pre-written code from classes that are not part of their own class hierarchy.

Implementing the ActionListener interface is equivalent to signalling to the Java compiler that you want a specific set of Event services. Part of the contract associated with that includes a commitment from you that your class will include the handler method actionPerformed. Java is therefore able to send the actionPerformed message to the objects that you have registered whenever the registered Event(s) fire.[119]

In a language such as C++, where one can arrange classes to inherit from more than one class (the multiple inheritance issue that we raised in Chapter 2), handler methods can be included directly – and you may be able to see why this idea of a Java class *implementing* an interface mimics that, without quite getting into what is still regarded as the murky waters of multiple inheritance.

Java has a number of standard interfaces. Many of them impose requirements relating to the attributes and behaviour of objects (as does ActionListener).

An important interface that does not impose such requirements is the Serializable interface. Classes implementing this are signalling to Java that their objects need sufficient metadata to permit them to be expressed as the sort of binary stream that can be sent down a wire. This is important in distributed object computing and we shall say a little more about it later.

[117] It is expressed entirely in the language of Java.
[118] Widget is a generic term for 'screen object'.
[119] Java does not include any way to associate a method with an address, so one cannot register methods by direct referencing, as one can in some languages.

© OUDCE, January, 2013

Building frames

We used Borland's JBuilder tool to mock up the following frame for the Porterhouse Library system. We claim no particular expertise in interface design and, as it happens, had never previously used the interface design features of JBuilder.

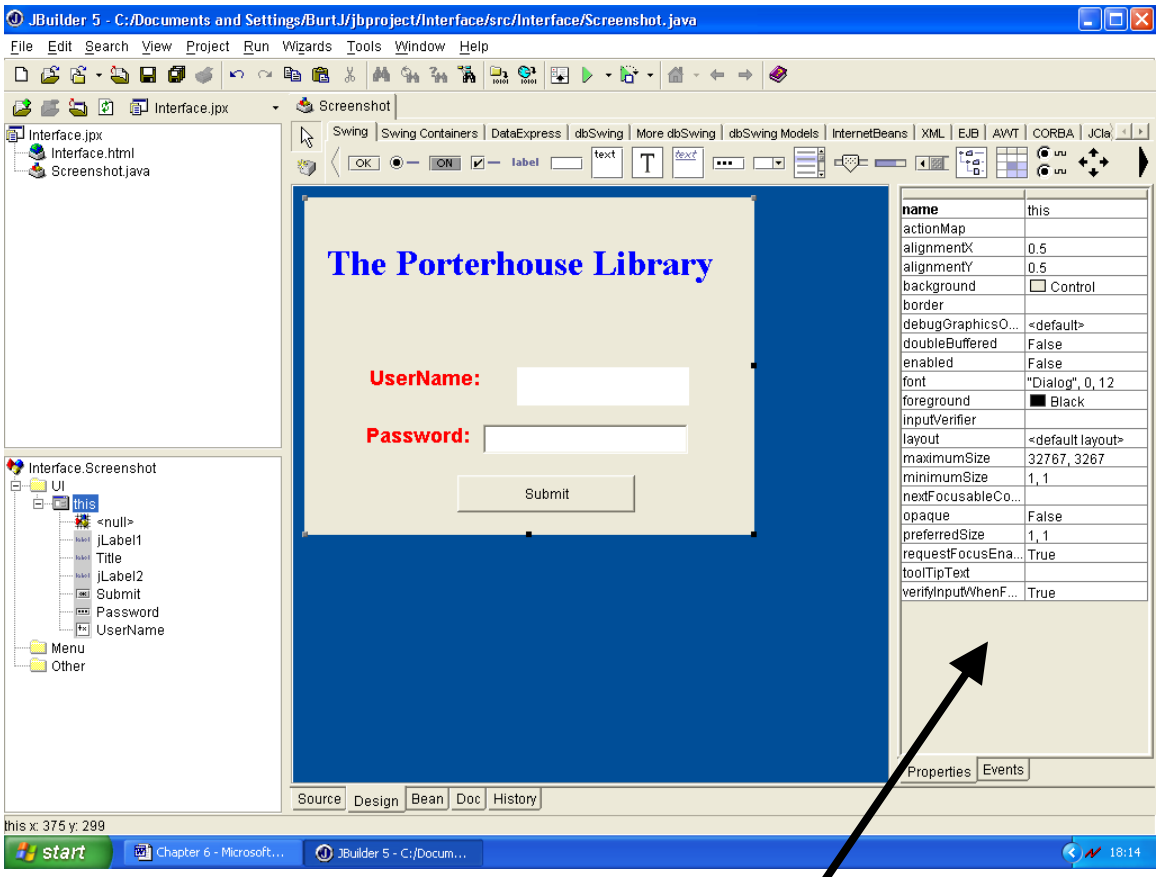


Figure 6.1 -Using the JBuilder Interface to Construct the Frame for the Porterhouse Library

You see the frame in the process of being designed[120]. The menu on the right-hand side of the screen lists properties of the displayed objects (that is, attributes). This particular frame contains three labels: a text area (for the user name), a password field and a button.

When we constructed this interface, we *used* a Borland IDE to *produce* an IDE. The Windows shown in Figure 6.1 are all part of this Borland IDE, and the middle Window shows the interface we are in the process of constructing.

There are similar facilities available in most development environments and for any languages that you are likely to use. Students of our classroom-based Advanced Diploma course routinely designed user interfaces for their projects over the course of an afternoon, working in Visual Basic.

Web interfaces can be produced either by hand, coding the HTML in a simple text editor, or by using one of the modern HTML editors.

[120] About your only experience of implementation in this unit!

© OUDCE, January, 2013

This kind of drag and drop editing does generate the underlying code for the frame widgets that you see. In languages such as Visual Basic, one can even connect Events generated by screen widgets to methods from the domain model using menus similar to the one arrowed on the right of our screenshot.

What happens in the system?

The idea is that the issue clerk's first encounter with the system is the frame shown in Figure 6.1. User name and password details are submitted and the submit button operated, which causes a message to be sent to the actionPerformed() handler method that Admin class must have[121] (because it implements the ActionListener interface and Java will not allow the code to compile without it). That method calls checkUserDetails() and that method systematically calls the user objects known to it, supplying the input user name and password details and asking for a match.

The user name and password details are attributes of the text and password fields in our frame. The information can be made available to any objects that need it –for example, by writing get accessors that return the information on demand.[122]

Here is an updated version of the sequence diagram given in Figure 5.4, recording our present understanding of what happens.

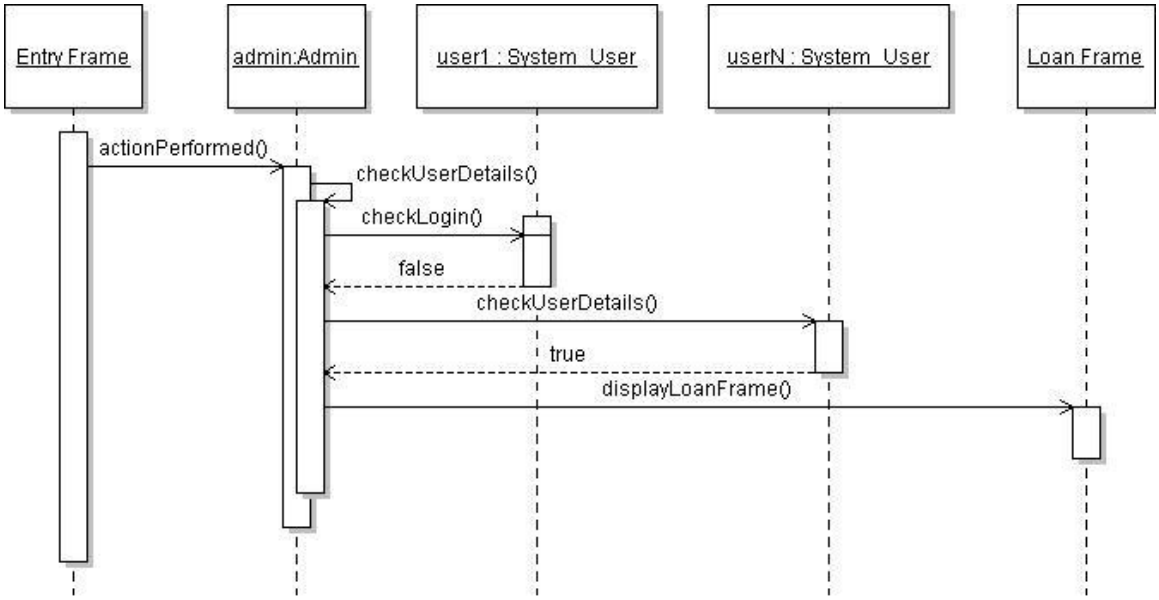


Figure 6.2: Logging in a Librarian

We have cut a few corners in this diagram because, as is standard with sequence diagrams, we have not shown all message replies, and we have not shown the parameters associated with messages. Like all sequence diagrams, this one relates to a specific set of messages associated with a specific set of objects.

[121] Java programmers will know that actionPerformed is often in another handler class that lies within Admin, a so-called 'inner class', but these details just complicate the story.

[122] Our stance is that attribute values should be obtained by accessors. In practice, in this sort of programming, objects tend to have references to frame objects and to read the values directly. They regard frames as local data structures, while slightly violating the principle of encapsulaton.

© OUDCE, January, 2013

Note that the checkUserDetails message is generated in the code for the actionPerformed method, and it is a message that admin sends to itself. (You should be able to make out the looping arrow indicating this in the second swim lane.) As before, we seek to represent what might be a series of many messages from admin to System_User objects, and we simply present the two alternatives. It should be noted somewhere that the message to user1 really represents a whole series of messages – to user1, user2, user3 … up to user*N*–1. Each of these messages would have had the reply **false**, indicating that the user name and password did not match those of that user.

We have introduced two interface frames into our diagrams. The idea is that we were constructing Entry Frame in Figure 6.1 and that the Librarian sees Loan Frame when trying to record a user loan. These ought to be proper system objects and questions of scope arise because our two frames belong strictly to the user interface and are not really anything to do with the system itself, that is, the **domain model**. You will see that we did not specify the frame names correctly and withheld class details. Scoping is also vitally important. You already know how important it is to decide on the boundaries of your system. Having separable user interfaces implies that you must be aware of an internal scoping issue relating to the division between the domain model of core code and the interchangeable interfaces that might access it.

The user interface does lie within the scope of the *project* because designers will have to produce it; although that process may be largely graphical.

In fact, the frame that we have identified as 'Loan Frame' is likely to be used for a number of different purposes (that is, unless Librarians only record loans, and never cancel them!). We shall probably have to change its name later.

We shall mention another interface frame in Chapter 9 – one associated with the privileges of the actor to which the Porterhouse Library specification refers as the Head Librarian. Our general design principle in this system is:

- Library staff fill in a frame, which asks them for a user name and password.
- They then see a further frame, which relates to the activities that their own particular user privileges permit.
- This frame may in turn lead to other dialog frames.

You see this clearly in Figure 6.2. It is often convenient for us to imagine that messages emanating from the manipulation of screen widgets by actors actually emanate from the system objects associated with those actors. In Figure 6.2, the key message was checkUserDetails() and that was sent from admin : Admin to itself. Technically, that message was certainly provoked by the manipulation of widgets in the EntryFrame, but that is simply an operational mechanism; the key object in all this is admin : Admin, not a frame.

We have glossed over user interface design by suggesting that one does it with mock-ups and storyboarding. We have to avoid saying much about this important topic because we simply have no space for it. You will certainly need to do some interface design, however, and to record your designs clearly. More than that, in some situations – for example, Web projects – the interface can be more closely associated with functionality than we have been suggesting so far, and this is at least partly to do with the fact that Web frames have such restricted inbuilt functionality – something to which we shall return in Chapter 10.

In describing a Web-based project one can achieve a lot with screenshots and text but we ask you to note our use of sequence diagrams above, which might also be employed to represent an interface based on hyperlinked web pages, with the objects replaced by web pages and the messages by hyperlinks. The moral is:

© OUDCE, January, 2013

You can use sequence diagrams to represent interface design.

It is possible to change interface frames dynamically on the basis of user needs or state. For example, one could arrange the display of frames with hyperlinked audiographic information for users who have problems with normal visual displays.[123] In such situations, domain model code makes decisions about presentation as the system runs. This sort of processing is very common in the modern Web.

The Porterhouse interface

The Porterhouse Library specification imposes non-functional requirements on the interface. It says that there must be 'an easy-to-use human interface for library staff with minimal training' and 'on-line Help facilities'.

The first of these must be achieved by the skilful use of HCI principles and experience, and by trying out possible interfaces with system users. The second can be achieved by exploiting facilities offered by frame software and by the production of textual documents that guide the system users through the things that are available to them.

We have illustrated the sort of facility that our frames could offer in the above hyperlink to the Porterhouse Library: if you are reading this as a Word document, and you hover your mouse over it, you should receive a bubble prompt of information on what it is and how you use it.

Design patterns

Separating interface considerations from the system is an idea that originated among people working in Smalltalk. It is an early example of a **pattern** – a recurring design situation that may be reused rather than developed.

Let us take a high-level look at the Porterhouse Library system. It involves:

- stock databases[124] (books)
- customer databases (borrowers)
- specialised staff who log into the system to make data changes relating to customers and stock

The system could describe a centralised system for distributing goods to outlying warehouses that could be employed by a major supermarket chain. It could also describe an Education Authority system used to control the allocation of equipment to schools. It could describe many things.

You will agree that, when viewed like this, our library system loses all claim to individuality and is recognised as just one particular example of something that is likely to recur in many guises. That being the case, we are perhaps foolish to attempt an idiosyncratic solution and might be better advised to modify a solution that already exists and is known to work well.

In recent years, increasing attention has been directed towards **design patterns**. A company implementing a system for the Porterhouse Library would be far more likely to use a pattern as a design template than to attempt the intricate and error-prone analysis associated with producing a totally new solution. The construction of modern software increasingly involves assembling a

[123] Information indicating that this is required is stored in database tables giving System_User details. The same tables might also say that printouts have to be in Braille, or whatever.
[124] We are using the term 'database' here in its broadest sense and do not wish to imply that relational databases *must* be used –though they *might* be.

© OUDCE, January, 2013

system from ready-made components based on the dictates of an appropriate pattern. This is particularly the case in distributed computing and we shall say a bit more about that later.

The major point is that the development of computer systems is becoming a component-based process.[125] The drag and drop facilities that we described in relation to user interface design are beginning to be applied even to domain model construction.

Model View Controller

Separating user interfaces from main or **domain model** processing is the basis of the **Model View Controller** (MVC) pattern.

The model	is the	central system.
The view	is the	presentation software.
The controller	is the	software handling user inputs.

This particular pattern seems appropriate to the Porterhouse Library, but we have also said, and we repeat here, that there are occasions when the interface considerations are central to system design and not to be relegated to the subservient role we give them here.

Chapter summary

Our analysis has given us some idea of what happens when people log into our system. In the next chapter, we shall go on to look at the ramifications of use case 1 – Issue Loan.

[125] In some sense, it always was. Programming projects invariably reuse or adapt solutions that worked on other projects. What is new in patterns is an attempt to codify that process.

© OUDCE, January, 2013

Chapter 7: Use Case 1

Library staff will most commonly access our system either to issue or to cancel loans. Here is what we know about this process so far.

1. Issuing loans is described in several places in the Porterhouse Library specification.
2. Our present analysis is that the transaction occurs using a special frame which we have tentatively called **Loan Frame** (see Figure 6.2).
3. This frame is displayed to actors only when they have supplied user names and passwords identifying them as members of the class Issue_Clerk (see Figure 5.2).
4. The system diary records that class Issue_Clerk corresponds to the specification term 'Librarian'.
5. It is unclear whether a 'loan' involves a single customer and a single volume or whether it could involve a single customer and several volumes.[126]

At this point we notice something that may well have occurred to you already – our terminology is loose. Look back to the candidate classes of our system (points 5.6–5.15 above). There, we use the term 'Users' to mean people availing themselves of library services, whether academic staff or students.

We have called the abstract class that describes the general characteristics of people who are going to log in to our system 'System_User'. Although we at least had the sense not to call it 'User' (and you might recall that we came close to doing just that), the term is still unsatisfactory and likely to cause confusion.

This emerges very clearly when we look at user case 1 because, as you will recall from Figure 4.4, an Issue_Clerk who has logged into the system must then 'locate user account', and this process will certainly resemble logging a 'user' into the system. We are in grave danger of manufacturing confusions through a poor choice of terminology.

You may find the occurrence of this problem exasperating after all the analysis we have gone through. On the contrary, it is fairly typical of analysis and design that potential showstoppers rear up unexpectedly, even quite late in the process.

We have two kinds of people:

- People using the system directly to provide services for others, who are presently handled as subclasses of an abstract class called System_User and relate directly to real system actors.
- People using the system indirectly to obtain services through the good offices of the first kind of people, who are presently described as 'Users'.

We have two *kinds* of people, but not yet, in the technical sense, two *classes* of people. So in what way does our second category of people correspond to the structures within the system?

The answer is that the system must somehow recognise these people as known to it and record their current borrowing and credit details plus contact details. This means that there must be classes describing the attributes and behaviour of these people and we had better give those classes names that do not make them liable to be confused with the classes associated with the library employees, who actually use the system.

[126] In point of fact, at the very start of the specification of the current, paper-based system, where it talks about *Borrowing Books*, there does seem to be an implication that a loan involves a unique volume. That implication is not reinforced in the subsequent discussion. See what you think, yourself.

© OUDCE, January, 2013

The specification mentions two sorts of people who are likely to borrow books (academics and students), but detailed analysis may well reveal additional categories, such as library staff themselves. Our suggestion is presented in Figure 7.1 below. You should be able to understand the UML now.

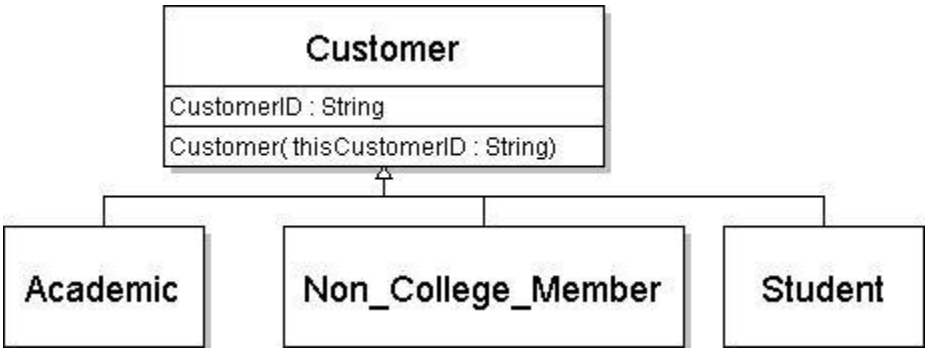


Figure 7.1: Classes Describing Library Borrowers

The diagram shows an abstract class called Customer and three concrete subclasses (though we may not actually need the middle one). Customers have IDs and, probably, a great many more attributes and methods.

We have contented ourselves with simply showing one method in this diagram – a constructor. The constructor creates a new Customer object which represents the Customer corresponding to the proffered ID.

You might have expected an interface method here, but constructors *are* in effect interface methods.[127]

Our present view of the system is that some objects, such as admin : Admin and the set of objects {user*N* : System_User ; with *N* = 1, 2, …} are present in the system at all times. although very many objects probably should not be. For example, the library may have thousands of volumes. Are we to expect our system to contain an active object corresponding to each one?

In this particular system, our Customer objects are created when needed and not archived: the implication is that they cease to exist when the Customer in question has obtained the services that they require (and, in Java terms, all references to them are voided).

Of course, some aspects of those services, such as loan and payment records, will have to be persistent. It could be that the objects *themselves* are stored in some sort of permanent way, and there are technologies to achieve that. If this is done, one will simply retrieve them from storage when they are needed and return them when they are no longer required.

[127] It is possible to make a constructor *private* in Java and you might like to speculate on the sort of circumstances in which that might arise, but constructors are generally public methods and can fairly be regarded as resembling interface methods in most respects.

© OUDCE, January, 2013

As designers, we face a technical choice, either:

- to arrange for the system to have persistent storage of objects – so that *objects* relating to customer accounts or volumes are retrieved from disk when required.
- to arrange for the system to have persistent storage of data – so that the *information* needed to construct customer account objects or volume objects is retrieved from data stores, perhaps by constructors, when needed.

The first solution is the more modern: it relates to our having some kind of object-oriented database associated with the system, or at least, to our ability to *serialise* objects.[128]

The second is the more traditional: it relates to our having some kind of database associated with the system.

We shall have more to say about the storage of persistent data, but you will now see that our library system is likely to need some databases.

Object-oriented databases

Object-oriented databases are starting to appear in commercial applications. It is probably fair to say that the IT community has mixed feelings about them and that their associated technologies are vastly less well understood than those offered by the relational model.[129] They are also expensive and involve skills that are not cheaply hired. On the other hand, the nature of object-oriented ideas makes some aspects of object-oriented databases very attractive and some of the things that one wants, viscerally, to achieve and which are difficult in relational systems – such as storing methods alongside data (because information transformation can be more complicated than is allowed in the flat transformational techniques of SQL), or handling many-to-many relations[130] – become easier to imagine in object-oriented contexts.

Vendors of relational software have responded to the object challenge by extending the sorts of things that their databases can handle seamlessly. You may have heard of CLOBS and BLOBS, which are exactly what one needs to handle in multimedia archiving and business applications.

> At the time of writing, designers would probably still go more for the traditional, relational solution to our problem. It would be cheaper, better understood and less likely to cause unforeseen problems. We shall assume that this is what we will do in what follows – and work with relational databases.

Another argument to support our use of relational databases comes from the functional requirements part of the specification of our case study: the Porterhouse Library. This talks of functions that search data using any appropriate search parameter. A relational database would give us that sort of capability *gratis* whereas other solutions are likely to involve us in expensive coding; although it must be said that modern object-oriented databases do support relational queries.

---

[128] We mentioned the serializable interface in Chapter 6. It permits a binary representation of objects, which can be stored onto a computer file or transmitted over a network.
[129] There is no widely-accepted object-oriented database model, although some influential suggestions have been made.
[130] Those of you who have studied relational databases will recall *intersection entities* and how counterintuitive and confusing they can be.

© OUDCE, January, 2013

Our nascent system handles fairly conventional sorts of data. Were the library to include multimedia objects – such as videos, audio graphics, or electronic data-sets[131] – an object-oriented database might start to look more attractive.

The constructor for Customer would probably use the database to read the following data.

- Personal information about *this* Customer: name, address, etc.
- Loan information about this Customer.
- Whether this Customer holds books that are overdue or subject to recall notices.
- Information on fines.
- Book requests involving this Customer.

The data is then stored in appropriate attributes of Customer objects. This information would be the basis of much of the decision-making that follows. Indeed, library staff will need to interrogate it and alter it, so the Customer object will require the appropriate get and set interface accessors[132].

When the transaction involving this Customer is complete, the data will be used to update the database. Perhaps the issue clerk will click a button called something like 'CONFIRM' and this will send a message to the Customer object, telling it to update the databases. The system references to this Customer object will then be deleted because the Customer object will no longer be active in the system. In C++ this can be done explicitly, by means of a destructor; in Java the story is more convoluted (see Chapter 2).

Our assumption is that the System_User objects of our system are present from the start. If we were coding in Java we could achieve this by having the static, main method inside the class Admin[133] and using it to create the admin : Admin object and all the System_User objects – storing references to them in a collection class attribute of admin : Admin. This initial processing would also involve accessing data from persistent database stores.

The Loan decision-making process requires information about the Volumes that are subject to borrowing requests. We could store references to every Volume object within admin : Admin, but this is likely to prove impossible in practice because there are just too many books! The required information is better obtained by directly accessing database records. The reader will understand that what we are describing amounts to an SQL query of relational data – and Java and other programming languages include facilities to make this an easy thing to accomplish.[134] We may not actually need to create objects corresponding to Volumes and Books because we can probably manage things better by writing methods that access database tables directly.

[131] We are thinking of the sort of data-sets that radio astronomy or particle physics might produce.
[132] Remember our account of them in chapter 2.
[133] Remember, this is the method that starts the system off.
[134] Be clear about what is happening here. The Java code would excite the database system into running SQL queries and returning the results; it would not access the data in the system directly - leaving the actual database operations to the database engine in question. You could change the database and nearly everything should still work (the database interface might have to change a bit).

© OUDCE, January, 2013

The Customer constructor may have some kind of additional 'software lock' that prevents other object instances that correspond to this Customer from occurring elsewhere in the system. This might be needed for security reasons, depending on how Customers identify themselves to library staff.[135]

Here is our account from Chapter 4 of the main success scenario (4.6) for use case 1. It comes from the tabular version of the use case description that we gave there.

7.1 Librarian logs into system.
7.2 Librarian locates user account.
7.3 Librarian locates volumes to be borrowed.
7.4 Librarian updates user account.
7.5 Librarian updates volumes database.
7.6 Librarian logs out of system.

Librarian is now called System_User and user is now called Customer.

We have already treated Step 7.1 to considerable analysis. Step 7.2 has been expanded. We now imagine that one establishes that this Customer is valid and then creates an object corresponding to it, using the constructor we mentioned in Figure 7.1.

The other steps are assumed to proceed without hitch because this is a success scenario, but we can isolate the (successful) checking that is done at each point.

Step 7.2 will include checking that *this* Customer has the ability to borrow more volumes. The Customer's allocation must not be exhausted and any money owed must be below a certain value. It seems that having overdue books does not prevent a Customer from further borrowings. You may remember from our discussion of Scenario 4.4 associated with use case 1 in Chapter 4 that we suspected that it might.

Step 7.3 will involve locating the library information relating to the Volume that the Customer wishes to borrow. In principle, it is possible to imagine that the book in question is subject to a reservation request, although the system described in the specification seems to say that books that are reserved are either removed from the shelves or retained as when they are returned. Our processing will probably be more robust if we include reservation checking at this point anyway, and that is what we propose.

Step 7.3 is poorly expressed. The volumes to be borrowed are, presumably, *physically* with the Customer.[136] What the step actually means is that information pertaining to these volumes is located. This is where we suggest that it is appropriate to access the database tables relating to library books.

There seems to be little point in constructing Volume objects here because, although Volumes are central to our work, we actually need very little in the way of behaviour from them, and most of their attributes are likely to be unchanged by our processing.

The moral is that an object-oriented system does not necessarily need to have objects relating to *all* the real world structures that the system models. Our list of candidate classes in Chapter 5 did

[135] This is probably a bit of a frill. If two students approached library staff at different terminals, proffering the same Customer details, they might potentially exceed book allocations. Perhaps this is impossible; we merely wanted to make the point that one *could* impose security controls. and that this is one place in which that could be done, since this capability is built into the database technology.

[136] This may not be strictly true in cases in which the customer has already reserved the book to be borrowed.

© OUDCE, January, 2013

identify Book as a possibility, but the functional requirements of our system seem to militate against our having system objects modelling books or volumes: to do so would be to add a complexity that is inappropriate. In our system, books and volumes correspond to entries in database tables rather than to active objects. Of course, other people might model the system differently or take a different view; and as teachers we have to say that we would expect students new to OO to *insist* on Volume objects. We would just not have them ourselves.

Steps 7.4 and 7.5 also need to be expanded. The local objects relating to customers probably contain information relating to current borrowings, and the library system certainly has to record that a given volume is out on loan. Since, however, by hypothesis, we are dealing with relational databases, we must be careful about storing information inappropriately.

> Consequently, Steps 7.4 and 7.5 are going to introduce entirely new design considerations into our system – the design and structure of an appropriate underlying relational database.

We expand on that in our next chapter.

© OUDCE, January, 2013

Chapter 8: Data

We hope that what we have said so far has taken us quite a long way into the process of analysing and designing an object-oriented system that will implement the requirements of the Porterhouse Library. In this chapter we shall look at some of the non-functional aspects of the system in more detail, in particular those relating to data storage.

Before we get to that, let us consider where we would be, had we extended our analysis of one particular use case to the other use cases of the system and completed the work that we began in Chapters 4–6:

- Our use case diagrams and descriptions would have given us an account of system functionality and scope.
- Our class diagrams would have recorded the main classes of our system and how they are associated.
- Our CARC cards would have given detailed information about the classes, their capabilities and how they implement system functionality.
- Our sequence diagrams would have recorded system processing at the message level, identifying messages, parameters and replies.
- We would have 'mocked up'[137] various interface frames associated with system functionality and tried them out on the people who will be using our finished system.
- We would have maintained a system diary that charted and organised the development process.

The point relating to trialling interface frames needs amplification for which we have little space here. There are established techniques associated with interface trialling, whose essential features relate to being systematic and recording user responses in a structured, usable way. (The response 'I don't like this!' is worth recording, but hard to work with!) We have to avoid discussing this here, but we do not wish to give you the impression that it is somehow unimportant.

We can imagine that at this point we have documented all the use cases and considered most of the scenarios, and that we have a fairly clear understanding of how the system behaves. We should also have a complete set of CARC cards for the classes in the system. We ought to be able to go through the system specification relating each functional requirement to a use case, and we ought to be able to conduct **structured walkthroughs** of use case scenarios, by issuing team members with appropriate CARC cards and role-playing what actually happens. Our CARC cards include complete information on interface methods, so this activity should be fairly easy.

A major flaw in our approach, so far, is that it has been almost entirely functional. We have worried about *what* the system does and our treatment of *how* it does it has been entirely predicated by functional concerns.

Consider, for example, what should happen when a power failure occurs or when the system needs to shut down? Does the system emerge with no knowledge of previous activity after such a shutdown? Clearly not!

[137] 'Mocked up' because we had not bothered to connect our interface frames to processing at this point. The frames would have been subject to change and the domain model processing had not yet been completed.

© OUDCE, January, 2013

Users in our system

Our analysis so far has taken us a considerable way into what happens when an issue clerk logs in, and one can probably imagine how volumes might be recorded as being on loan or returned.

The interface we have described covers a familiar situation: a single computer addressing some sort of data repository. It is not entirely clear, however, that this is the system that is wanted.

One could normally expect that our final statement of requirements would address this issue as a non-functional requirement. What are the options?

Multiple access options

8.1 The library is small and access involves a single PC which holds all the data.
8.2 The library is big enough to have several PCs accessing data managed by a network operating system.
8.3 The library is very big – occupying multiple sites – and remote access via the Internet is possible.

The Porterhouse Library specification suggests that the system should initially consist of a single PC (Option 8.1), which is the simplest option. Only one person can be logged in at a time and there is no need to worry about consistency issues. Option 8.1 corresponds to the Rewley House library system, or how you might manage a system recording your own CD collection.

That worrying word 'initially' occurs in the specification, however, and one would want to consult with the commissioners about it. One might well end up having to produce something rather more complicated if that tiny word indicated the quaint belief that a single-solution system could run immediately on a network.[138]

Option 8.2 is more realistic. It probably corresponds to the situation in the Oxford University History Faculty library (which occupies several floors of a large building). It could cause us to worry about the 'same user'[139] logging in simultaneously from two different locations and abusing the consistency of the library data. The risks associated with that are minimised in this project because the users of the system are library employees rather than members of the college. We might still be inclined to put in software checks to prevent the 'same' librarian from being in the system 'twice', and you might like to consider how this might be done, but the issue is not likely to be very important.

A more serious problem arising from Option 8.2 is that two library users armed with the same user details might attempt to obtain services from two different issue clerks simultaneously. User services are covered in the Porterhouse Library specification, but that document is vague about how students identify themselves, and it even talks about books being returned 'anonymously'. Identification at Oxford University libraries is on the basis of a University Card. Consequently, multiple occurrences of the same user should not be possible.[140]

If we accept the *possibility* of multiple occurrences of the same user within the system, what can go wrong?

---

[138] When customers are not telling you that they know nothing about computing, they are making authoritative technical assertions! Requirements elicitation involves diplomacy and patience and should not be undertaken by the easily stressed!

[139] So far as the system is concerned, a user is the conjunction of a user name and a password. It is possible to imagine this information being proffered at two different terminals, because human beings sometimes share sensitive data.

[140] I have obtained library services, however, simply by looking appealing and without having to show a card! At least, that is how I thought I got them! This is a serious point – people don't always behave as specifications suggest.

© OUDCE, January, 2013

The Porterhouse Library specification covers a number of library services for users, among which are (using the language of the specification):

- Borrowing books
- Returning books
- Reserving books
- Paying fines

The first two of these involve the physical presence of what we are calling Volumes. A Volume cannot be in two places at once so most of the problem is solved. It might be possible, however, for a student with 19 books on loan (one less than the maximum) to hack the system by soliciting a friend to borrow a Volume in the first student's name at one terminal, while he or she borrowed another Volume in his or her own name at a second terminal. That might in turn introduce the possibility of a 'lost update' problem, with Volumes being borrowed but not recorded in Loans.

Option 8.2 might cause us problems with the third service. One might imagine distinct users at distinct terminals reserving the same book.[141] It looks as though some sort of **data locking** is going to be necessary there.

A similar problem might occur in paying fines. Again, it seems that we shall have to lock user data.

The problems with Option 8.3 are of the same sort, but worse! The solution to all these difficulties is suggested by some of the other requirements of the system that relate to searching the information on Borrowers. This looks like a database query, and the suggested solution is to store data in **relational databases**.

Feasible and appropriate solutions

It is certainly possible to imagine a pure object-oriented solution to producing a Porterhouse Library system; but the above discussion, the previous discussion in Chapter 7 and the part of the specification involving data searches tend to make such a solution inappropriate so far as present-day technology and easily available skill-sets are concerned. There is no point in delivering a system whose maintenance requires expertise that is not readily available on the open market or techniques that are still being developed and are likely to change radically in the immediate future.[142]

We assume that you have a basic understanding of relational databases and would be capable of designing one that meets the needs of the Porterhouse Library. We suggest the following database tables (which you should compare with the list of candidate classes 5.6– 5.16).

- Book
- Volume
- Customer
- Loan

You ought to be able to sort out primary and foreign keys to link these tables. For example, there is a one-to-many relationship between Book and Volume; and Loan has a one-to-many relation with Volume and a many-to-one relation with Customer. This multiplicity slips a design decision past you, however, which we hope you noticed! We ruled that a customer loan involves a single Customer borrowing a single Volume, because it makes the processing relating to returns and

[141] Recall that one reserves *books* and borrows *volumes.*

[142] The sort of techniques we are talking about, involving distributed objects and object-oriented databases, are available now and increasingly coming into use.

© OUDCE, January, 2013

reservations easier to handle. If one Loan consisted of the Volumes that a Customer wanted to borrow at one request, then handling book returns in which only some of the requested Volumes appeared together would be more difficult.

> A customer attempting to borrow three Volumes will appear in three new entries in the Loan table.

The system we would deliver would have a relational database with these tables. The original Porterhouse Library specification speaks of a catalogue of library stock and requires functions to manipulate that catalogue. You will see that all these things can be done very simply by using a relational database and that, more important, they will not require us to do any complicated coding because they correspond to well-understood SQL statements. More than that, the fallow period when Volumes may not be loaned due to the need to validate the stock list may not need to occur at all with this solution!

We could probably handle the specialised functions in our system – the ones that relate to maintaining the Books and Users database – purely by means of frames linked to SQL processing. This approach gives us transactional integrity handling as a bonus, plus the possibility of offering fairly sophisticated functionality rather quickly. Also, by having the tables that we listed in 8.8–8.11, we can also offer such things as loan archives and user choice targeting.[143]

Objects remain a big feature of our system; but permanent data storage involves relational databases and the objects of our system do not have to map directly to our database tables.

Our database

The database we constructed for this system has the following tables:

1. Author
2. Book
3. Volume
4. Customer
5. Loan
6. Reservation
7. System_User
8. Publisher

Of these, Tables 2, 3, 4 and 5 relate to Use Case 1, and the other tables relate to other services that the library is supposed to provide.[144]

We have the relationships as shown in Figure 8.1. We hope that you can make out the relationships and keys. If you have Microsoft *Access*, you can see the database we built, but that would be just for fun; you don't need it for this unit and you can probably improve on our design anyway! (See Porterhouse Library Database.)

[143] We could alert users with specialised interests to new acquisitions that they might find useful, and we could even do this on the basis of their previous loan activity. I am reminded of the horror film “Seven” in which the suspect was tracked by searching for people borrowing specific disturbing books at public libraries (probably on object-oriented analysis).

[144] We should confess that Publisher table is a bit of a frill in terms of the system specified!

© OUDCE, January, 2013

Observe that in our database:

1. The Loan table has a multi-field primary key including: Customer_ID, Volume_ID and time and date.
2. Each Volume borrowed produces another row in the Loan table.
3. The Reservation table has a similar three-field primary key, but this involves the ISBN rather than Volume_ID. (You borrow *Volumes*, but reserve *Books*.)
4. System_User is a table defining the people who can log into the system.
5. System_User has no connections with the other tables.
6. Customer does not have user name and password fields because Customers do not log into the system. They identify themselves using their Customer_ID (perhaps bar-coded onto their university cards) and that is the primary key for the Customer table.
7. The Account Value field in Customer relates to debts owed to the library.
8. One Book can exist as several Volumes.
9. We don't have a reason for including Publisher that would stand up in court. The specification does talk about stocktaking, so we assume that the acquisition of new stock is of some interest.
10. The fact that System_User table is not related to any of the other tables might trouble people who are used to database handling. The table relates to privileged operations within our system and those who can perform them. Isolating the table gives us a small measure of additional security Also, as you are about to see, the table is used at system start to generate System_User objects, which are referenced from Admin[145] class.

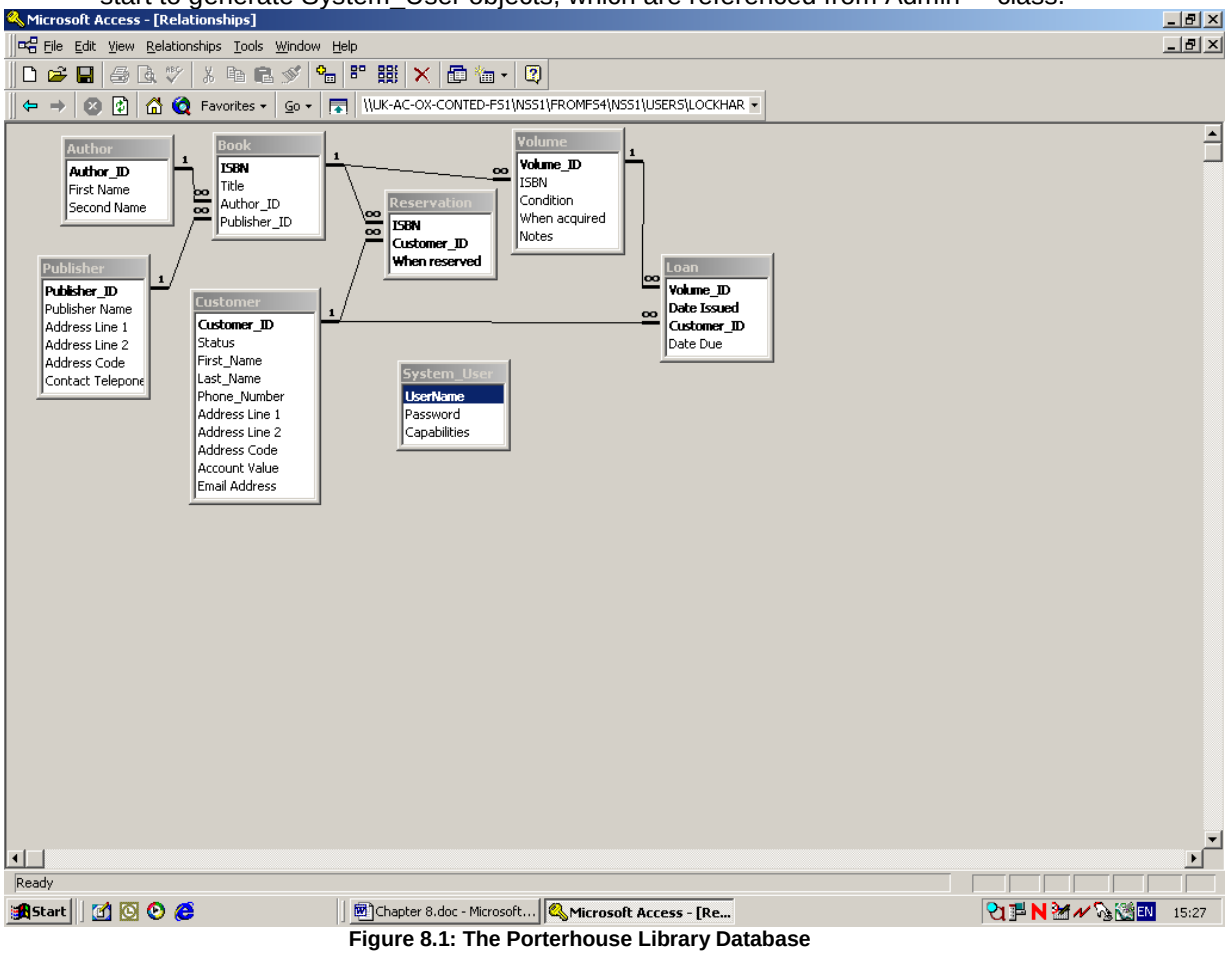


Figure 8.1: The Porterhouse Library Database

[145] The code we will show you constructs static references within Admin class, but we are cutting corners and it would be better to use instance variables in admin : Admin.

Database access using Java

We shall now outline how Java might be used to access the database tables that we have described, but we shall not give you the full story. What we tell you is intended to be purely illustrative: you do not need to learn, understand or remember it! You will find a much more authoritative and comprehensive account of database access in unit three, we include one here merely for completeness. You will similarly find that unit to be a much better reference for information about relational data, so don't be afraid to refer back to it when you read what follows.

Departing from use case 1, we shall consider the start of our system. Then, by hypothesis, the Admin class starts things off by creating its single admin object and then creating objects relating to possible System_Users. We shall outline the creation of these System_Users.

The problem is to access a database using Java. We would start by making sure that our code can actually locate the database. On Windows systems this would involve the Control Panel, and probably the ODBC data source icon that is often found as part of the set of Administrative tools.

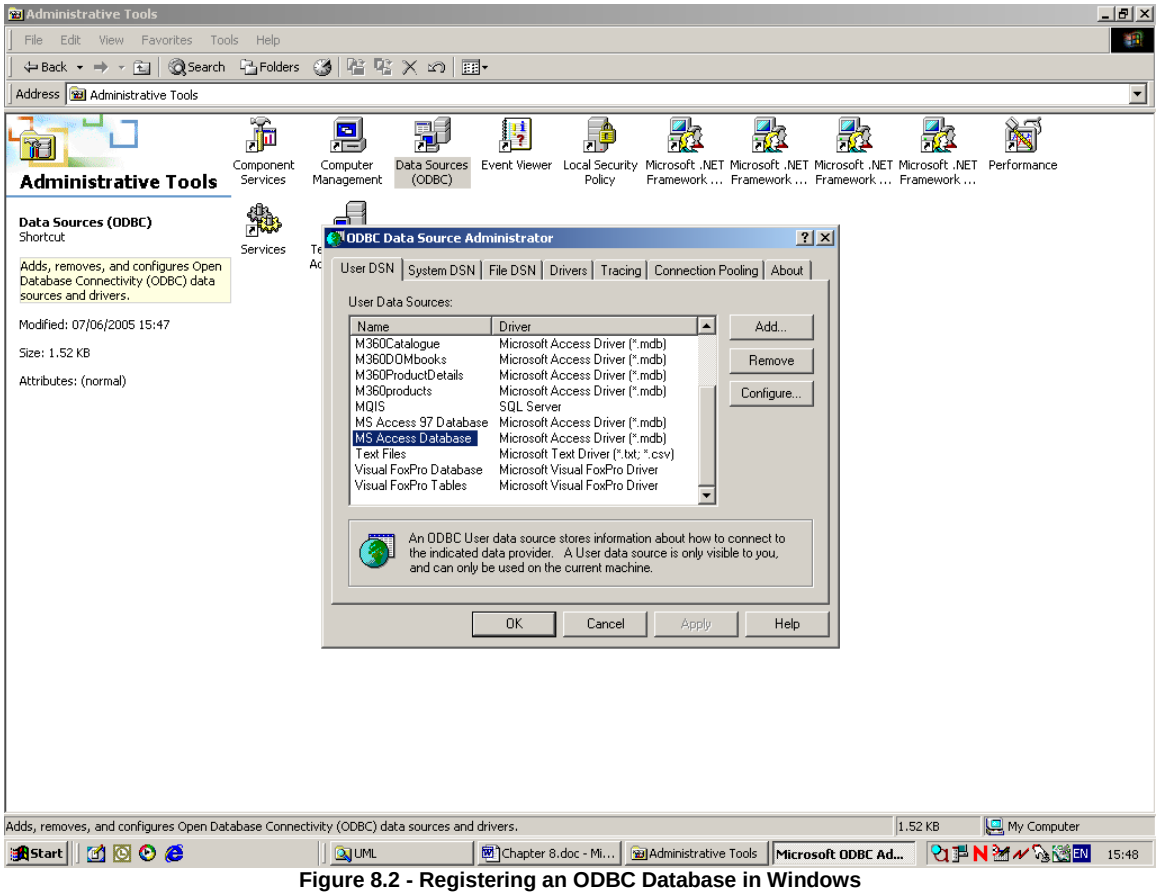


Figure 8.2 - Registering an ODBC Database in Windows

The idea is that bridging software connects the Java program to the database engine and translates from Java into something the database understands. The key to the whole thing is SQL and you will see SQL instructions in the Java code that follows. The actual connection from Java to the database uses Open Database Connectivity and the screenshot in Figure 8.2 covers registering an ODBC database source with the operating system.

One then writes Java code to access the database. Java is a book-keeping sort of language, so you must expect a lot of extraneous form-filling in the code that we show you. Just concentrate on the main ideas: we annotate them and, if you are reading this electronically, our comments should appear in red. This program is illustrative, so just get the gist of it.

© OUDCE, January, 2013

```
import java.sql.*; //Makes the package of database access code
                   //available.
public class Admin  //we won't show all the aspects of this class,
                    //and merely concentrate on using it to access
                    //a database table.
{//start of the class definition
     //we first declare three vectors that are going to store objects
     //that represent Issue Clerks, Head Issue Clerks, and System
     //Managers (see Figure 5.2).
     private static Vector Isue_Clerk,
                           System_Manager,
                           Head_Issue_Clerk;
                  //used to store System_User objects
     public static void main(String[] args) //the standard start
                                            //to a Java program
     {//start of main method
          //This sets up the character Strings needed to make the
          //connection to the database.
          String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
          String protocolHeader = "jdbc:odbc:";
          String dbName = "Porterhouse Library Database";
          String user = "BobL";
          String password = "LovelyStudents";
          try  //this illustrates how Java lets you catch exceptions.
               //When you are running this sort of connection code a
               //lot can go wrong so you need to watch out
               //for run-time errors.
          {
            // Loads the special Java classes needed to handle ODBC (an
            //(exception is thrown if the classes cannot be located).
            Class.forName(jdbcDriver);
            // Connect directly to the database
            Connection cn = DriverManager.getConnection(
                 protocolHeader+dbName, user, password);
            //If something goes wrong here, an exception will be thrown
            //and picked up in the code that starts with the word CATCH

           //Now the code that creates an SQL query which Java invites
           //the database engine to run on our behalf. We don't expect
           //that you are going to know a lot about SQL -
           //just get the idea
            Statement query = cn.createStatement();
            //now read the small System_User table
            //first, set up the SQL string.
            String system_user = "SELECT * FROM System_User";
            //next, execute the query (note, this can throw exceptions
            //too, and they would be caught below)
            ResultSet rSet = query.executeQuery(system_user);
            //Notice the textual nature of this high-level query
            //When we get to here, we have returned from accessing the
```

© OUDCE, January, 2013

```
            //database

            //Now process the individual rows in the table sent back
            while(rSet.next()) //we go a row at a time
            {//start of the while loop. The getString method
             //returns the database field entry as a String
            get_New_User(rset.getString("UserName"),
                         rset.getString("Password"),
                         rset.getString("Capabilities"));
//Here, we are assuming the existence of a method called
//get_New_User which takes the three string parameters read from the
//database and decides what sort of user to create (on the basis of the
//settings in Capabilities) and then calls an appropriate constructor -
//storing the result as an object in one of the three Vectors used to
//hold System users.

         }//end of the while loop.
         catch (Exception e) //This is the code that handles the
                             //run-time exceptions associated with
                             //failing to access the database.
             {
                 System.out.println("We could not connect to
the database. The error is: "+e);
             }//end of exception handling
     }//end of main method
}//end of the class specification (don't forget we left a lot out!)
```

This code connects to the Porterhouse Library database and reads all the entries from the System_User table.

This table has a field called Capability, which determines whether the System_User is an Issue_Clerk, a Head_Issue_Clerk or a System_Manager.

These three kinds of user correspond to three related classes in our system (see Figure 5.2). Our code imagines that the Admin class has a static method called get_New_User which takes the three attributes of the System_User table – Username, Password and Capabilities – and calls appropriate constructors to produce objects that represent the appropriate kind of System_User. The method would then store the newly created objects in one of the three static Vectors used to hold them.

In fact, the code would almost certainly not work in the way we have described. It is more likely that the main method of Admin class would start by building the single admin : Admin object in the system (that is, it would call its constructor) and that *this* constructor would handle the processing, including the database access. The three Vectors shown would then be proper attributes of admin : Admin, rather than static class variables.

The above code illustrates quite a few of the programming ideas covered in Chapter 2, and it also shows how Java database accessing works. We hope that you do understand it in outline, but there is no need to worry if you do not and as we have observed, you will find a far better treatment of these ideas in unit three.

© OUDCE, January, 2013

Addendum

The previous notes relate to the installation of a JDBC-ODBC bridge, which is a particular sort of database driver. To the best of our belief, everything reported here is correct, but the situation has changed a bit since these notes were originally written.

These days, Java access of databases is much more common and some of the hurdles that used to be in place have been removed.

You can get an up to date story on how to proceed from the SUN website (though as we mentioned above, this material is included here only for completeness and nothing to do with the main thrust of this unit).

The modern story relating to program access of data sources involves two Java classes: **DriverManager**, and **DataSource**.

These classes provide the interface you need. DriverManager will seek out JDBC drivers, loading the ones it finds (though older drivers will have to be explicitly loaded). DataSource is, perhaps, more modern. It insulates you from much of the detail of making a connection. You really only need a URL pointing at the database in question and that can lie on your own machine – SUN give the example: **jdbc:mysql://localhost:3306/**. Here, localhost identifies the server hosting your database and 3306 is the transport port used.

What this addendum is really saying is that it's actually easier to connect to databases using Java than it used to be. You can find more information on the website:

http://download.oracle.com/javase/tutorial/jdbc/basics/index.html

© OUDCE, January, 2013

Chapter 9: Further Issues

What we have said so far has taken us a considerable way into the design of a system for the Porterhouse Library and covered a number of key object-oriented ideas.

The system we have been describing is something of a hybrid. Our goal is to give you an idea of how objects are actually used in modern developments and for that reason we did not construct an artificial solution that was entirely object-oriented. Indeed, some of the candidate classes identified in Chapter 5 were not actually implemented: we found no need for Book or Volume classes because these concepts seemed best realised as entries in relational database tables.

State Diagrams

Here is a **state diagram** for Volume objects –for the hypothetical case in which our analysis had not ruled them out!

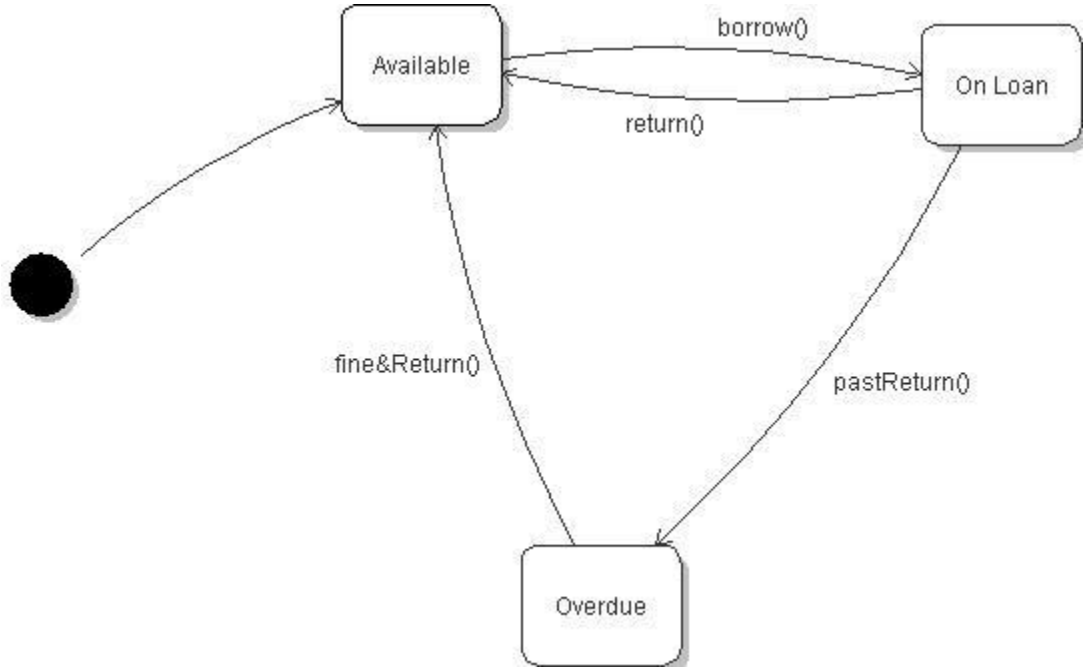


Figure 9.1: State Changes for Volume Objects

The diagram shows that Volume objects start in the Available state (with the black dot indicating the start of the object's life). They convert to the On Loan state as a result of the borrow() message, and from that state they may change into the Available state or the Overdue state.

State diagrams represent the state-related behaviour of an object and how they react to key messages. This one would not relate to real objects in our system because we simply do not have Volume objects; and if we did have Volume objects, we would probably want there to be further states relating to Volumes being withheld from loan and Volumes being reserved.

© OUDCE, January, 2013

State diagrams have the following constituents.

1. An optional symbol indicating the initial state. This is the same symbol that was used for this purpose in activity diagrams, which we show above.
2. An optional symbol indicating the final state. –This is not shown above because it is not appropriate.[146] Again this is the same symbol that was used in activity diagrams.
3. States, shown as named rectangles with round edges. There are three states in Figure 9.1.
4. Arrows indicating state **transitions**, which are associated with **events** (see below). In object-oriented systems these usually correspond to messages. State transition arrows are often labelled with the corresponding message, as is done in Figure 9.1.

Although events normally correspond to messages, they sometimes relate to other things, such as human actions, which generate so many messages that it can be useful to use event names rather than messages in arrow labelling. An illustration of this in the Porterhouse Library is when a customer brings in an overdue book, which might spark off a whole sequence of messages to a whole range of objects.

State diagrams are useful documents in systems with objects that undergo clear state changes which correspond in turn to behavioural changes (state-related behaviour). They are closely related to activity diagrams but focus attention on the state complexity of objects rather than the flow of activities. The Porterhouse system we have been describing probably has no objects with the kind of complexity that would require us to use this technique very seriously.

Here is a state diagram relating to a bicycle tyre.

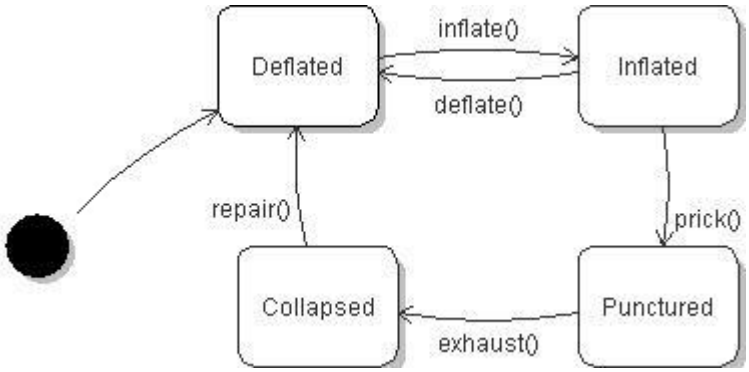


Figure 9.2: The States of a Bicycle Tyre

This little diagram distinguishes between Collapsed and Deflated states (where Collapsed is associated with injury to the tyre). One can get to the Inflated state from Deflated, but not from Collapsed. You might argue that one should be able to get from Deflated to Punctured – and this might be a desirable thing in the system one is building; but whether one can or not depends on the code and design of that system and not, directly, on the properties of bicycle tyres in the real world. In the real world a tyre would occupy a continuum of states between Deflated and Inflated[147]. One could, of course, model that, for example by having a floating-point attribute called pressure. This is not done here.

[146] It is possible to imagine that battered Volumes are withdrawn from the library, perhaps sold off – and that such Volumes might have some final, purgatorial state.
[147] Well, perhaps not my bicycle.

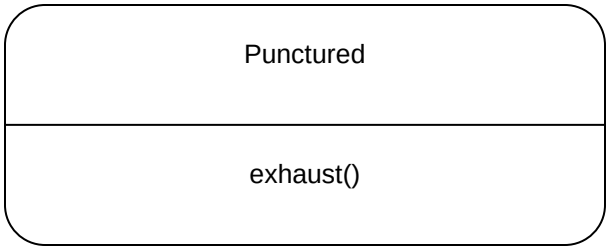
© OUDCE, January, 2013

Software models are abstractions: they may represent features of real world situations but they never duplicate them exactly and there is no suggestion that this is ever possible.

A recurrent problem with object-oriented systems is that Class names can suggest real world constructs that turn out not to behave quite as the programmer expects. Beware of subconscious prejudice luring you or your team members into false assumptions.

Indeed, some purists argue that one could take this issue even further: that designers should not bring *any* preconceived ideas about how, in our case, a library system works, but should rely entirely on what the commissioner's request. You might like to consider this issue yourself. Our view is that it's fairly silly to imagine that requirements elicitation ever takes place in some abstract setting of pure functionality.

States can be associated with **actions**. In the world of objects, this would usually mean that the associated objects send messages on entering the state. In the example above, the nature of the Punctured state is such that the exhaust() message is sent automatically. One might represent that like this.



There are also ways of representing actions provoked by state changes in the labelling of the state transition arrows.

The Punctured state is **transient** in that, once entered, it automatically transforms into the Collapsed state,[148] that is, it is transformed without further stimulus. We make no statement about how long transient states take to change. This depends on their nature.

In the Porterhouse system, Customer objects are created when needed and then destroyed when the transactions involving the Customer are at an end. In C++ this might involve **destructors** but in Java the whole thing would be achieved by de-referencing the object and automatic garbage collection (see Chapter 2 for details).

However it is achieved, Customer objects have *initial* and *final* states. One might *represent* this with the following state diagram.[149]

[148] Something that usually takes our students the whole period of the course!
[149] We have capitalised the Construct and Destroy messages because they are not really interface messages of the object (they are more to do with the class and the system).

© OUDCE, January, 2013

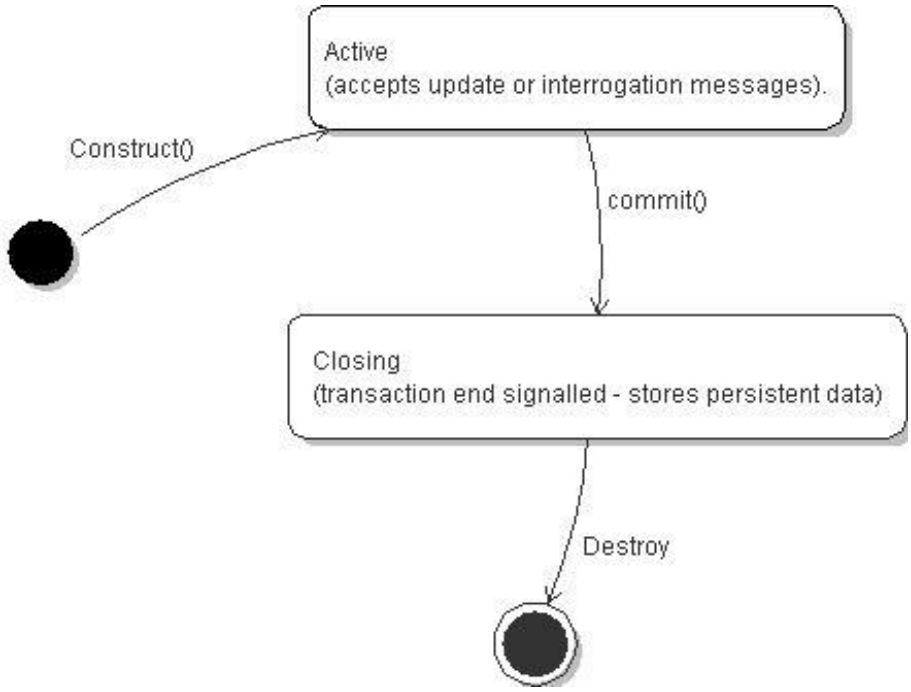


Figure 9.3: States of Customer Objects

Customer objects are in the Active state when they can accept interrogation or update messages. No assertion about system state is made; it may be that we could not in practice send some or all of these messages.

Customer objects moved to the transient Closing state when librarians end transactions with them. We have not yet specified how this is done but one could imagine a button marked **End Transaction** that would stimulate admin : Admin to send the commit() message.

The Closing state is transient. The Customer object updates the relevant database tables, unlocking any that are locked, and then reports back directly to admin : Admin in reply to its commit() message.

admin : Admin then deletes the Customer object. Although we have shown that as a message, you know that this would probably be achieved by other means.

The closing transient state is maintained while the relevant databases are updated. You might like to consider what happens should these updates fail: it may be that the system is small enough to permit the data to be stored to a file for human intervention; or it may be that the activity in question is doomed to be abandoned.[150]

You should be able to see that state diagrams *can* give you direct and useful information about objects. More than that, if they are annotated sufficiently, they can even be used to convey useful information about the mechanics of your system when the objects they describe do not really exist (and Figure 9.1 is an illustration of this). This can be helpful as a high-level design tool or to clarify policy, but you do need to ensure that your readers are not confused about what you do or do not implement as objects.

[150] That said, it is difficult to see how you could refuse Customers the right to *return* books simply because your system had some problems.

© OUDCE, January, 2013

Figure 9.1 could be a useful diagram at the system specification phase because the commissioners would be able to interpret and criticise it. It might influence decisions made during systems analysis; but it won't come into system design at all because there is no Volume class. If, at some future stage of the development, a Volume class became desirable, Figure 9.1 might influence how it was built.

State diagrams are a sort of object-oriented version of entity life history diagrams,[151] and they are one of the diagrams that *Violet* offers.

If you understand Figure 9.2, you should be able to answer the following questions.

1. In which state do bicycle tyres start off?
2. What message should be sent to a bicycle tyre in state Inflated in order to change the state to Punctured?
3. Can one get from the Punctured state to the Inflated state directly?

However, you probably could not answer this question.

What happens when a tyre is in the Punctured state and it receives the message: inflate()?

We shall now deal with this question.

Errors and exceptions

In Chapter 2, we mentioned that object-oriented languages tend to be fairly robust, and we said that this makes them very suitable for distributed applications.

Some part of that robustness comes from the way that modern object-oriented languages allow programmers to respond to run-time error conditions – that is, to **exceptions**.

We slipped in some Java exception handling in the code in Chapter 8. If you look again at the **try … catch …** construction in that code, you will see that noted in our annotation.

No programmer writes code in which errors are likely to arise but most complicated programs are at the mercy of situations that are difficult to predict or control.

A rather trite example of this might occur in a program that solicits a user to enter two numbers, and which replies with the number obtained by dividing the first by the second. All goes well, until the user enters zero for the second number. Any programming language would flag an error when division by zero was attempted, that is, **a run-time exception**.

The code in Chapter 8 involves loading special classes that may or may not be present in the system, and then attempting to connect to a database and run SQL queries. Anyone with any experience of databases will know that this is a potentially fraught process, and one that is likely to go wrong in a wide variety of ways.

It is true that one cannot possibly cater for *every* error that could occur, but experience recognises run-time errors that are recurrently likely (such as division by zero as we mentioned). Java and other object-oriented languages respond to such possibilities with the idea of an **Exception**.

Consider Figure 9.4, which shows a simple Java program in which division by zero is attempted. Java has responded to the attempt to divide by zero by creating an **Exception object**; the virtual machine terminates processing when it sees it.

[151] We appreciate that you may not have come across entity life history diagrams.

We hope you can just make out the error report. It says:

java.lang.ArithmeticException: / by zero
at tryout.Tester.main(Tester.java:9)
Exception in thread "main"

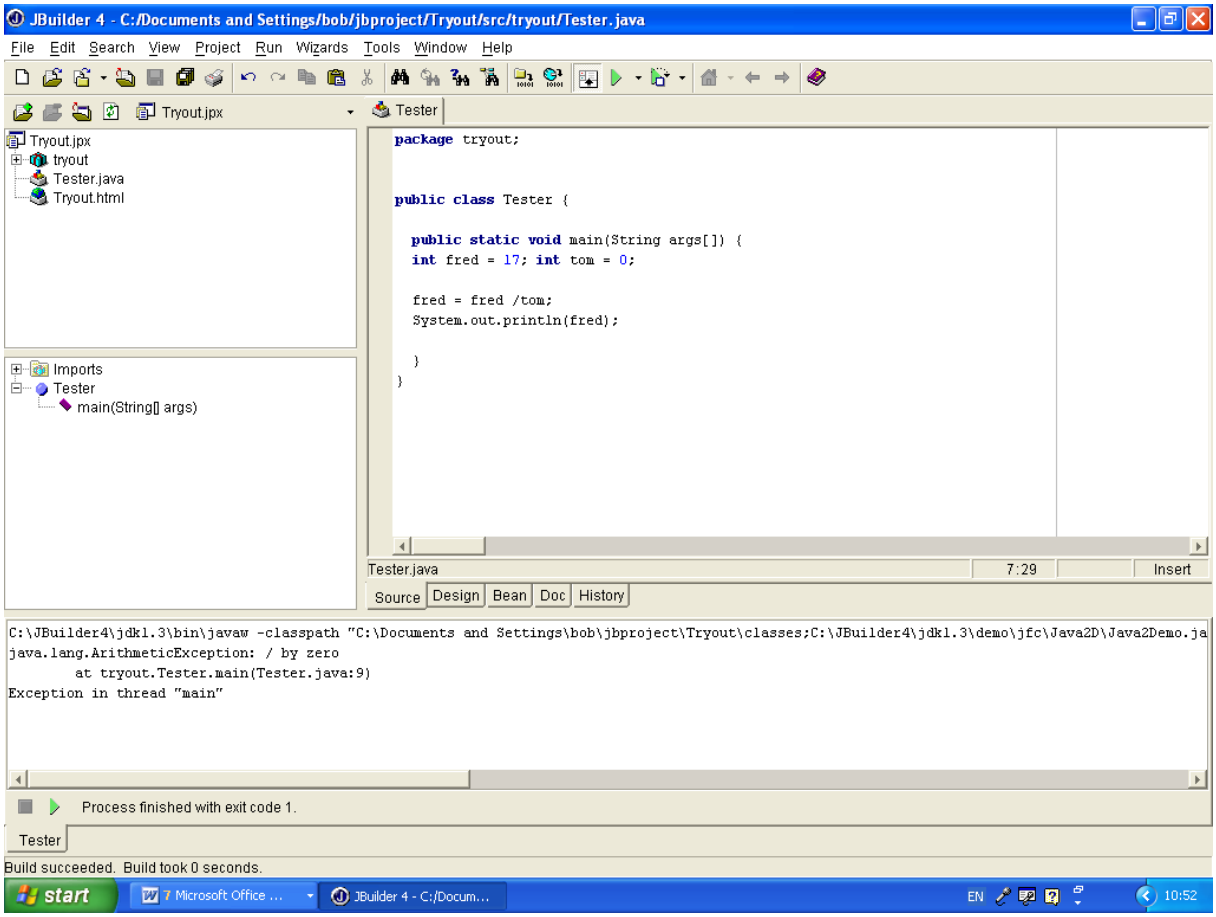


Figure 9.4: Division by Zero in Java

However, the code *could* have anticipated this particular exception and catered for it. Here is how we do that.

© OUDCE, January, 2013

```
public class Tester {
     public   static   void   main(String
     args[]) {
          int fred = 17; int tom = 0;
          try{
//java might throw an exception when the division is
//attempted. This would prevent the assignment statement
//from occurring and take us immediately to the catch code.
          fred = fred /tom;
          System.out.println(fred);
     }
     catch (ArithmeticException e)
     {
          System.out.println("Don't
          make tom zero, please!");
     }
}
```

We have attempted to show the try code in blue above. It signals to Java that exceptions are possible. The **catch** code (shown in red) stipulates what should be done if an ArithmeticException occurs (as it would for division by zero).

When the division is attempted, Java **throws** an ArithmeticException object, which is **caught** in the catch code and the message 'Don't make tom zero, please!' is displayed. Notice that the exception is thrown when the division is attempted and no more of the code in the try section runs. Here is what this looks like.

© OUDCE, January, 2013

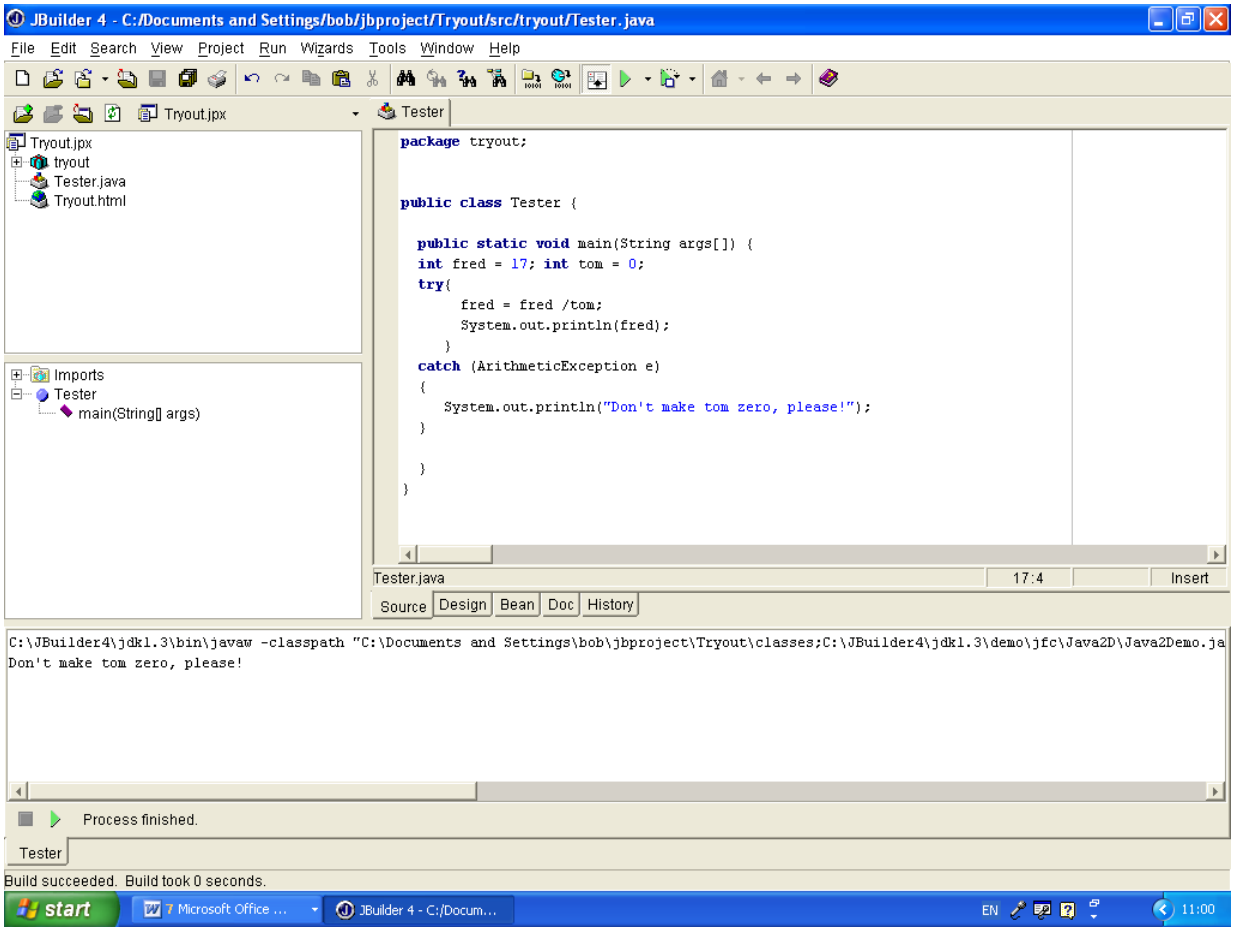


Figure 9.5: Catering for an ArithmeticException

This code only caters for ArithmeticExceptions. Any other exceptions that are generated, but not 'caught', would cause processing to halt.

We might say here that it would be a profound blunder to catch an exception but then do nothing at all about it! That sort of sloppy coding lay behind the crash of the Ariane 5 spacecraft.

We can now return to our earlier question:

What happens when a tyre is in the Punctured state and it receives the message: inflate()?

We should first observe that it is integral to the way languages like Java and C++ work that the tyre *could* receive the inappropriate message inflate() when in the Punctured state. Popular object-oriented languages do offer state-related behaviour but not state-related message handling.[152] So, it is theoretically possible for this message to be sent, and we therefore have to deal with it.

[152] It would be possible to imagine a language in which some messages were only transmitted when the receiver was in a set of designated states. That would not solve our problem, however, since it would merely translate the error handling to the sender rather than the receiver.

© OUDCE, January, 2013

Options for error handling

- One option might be to decree that we preface each message that relies on a given state with a previous message to check that the state is appropriate. This won't work, particularly in distributed environments, because a state change might occur between the first and second messages. One *could* decree that the state-checking message causes the state to freeze until the second message is received, but this is a rickety solution[153] and not in keeping with how object-oriented and distributed computing ought to work.
- Another option might be to decree that our system is so well defined and robust that these errors cannot possibly occur. This is often done in real life,[154] but it seems suicidal for distributed applications or applications involving databases.

Errors are likely to occur in several places in our system. Indeed, some of the Java APIs, such as the JDBC one associated with accessing databases, presuppose exception handling from the programmer; although with databases the best thing to do is often simply to report the error as clearly as possible, because it might relate to database settings that will have to be changed before access is possible. The code in Chapter 8 involved this very weak exception handling because there was no appropriate alternative.

Given that one regards run-time errors as important, and that the facilities one wishes to use sometimes insist on exception handling, it becomes important to have a unified policy about how they are to be handled. The key points seem to be the following:

- Errors should be reported clearly, including good information about how and where they occurred.
- Error reports address two classes of people – system maintainers and system users.
- System users need to be given help regarding whether or not the action they are attempting is appropriate or possible and how they should proceed.
- It is often rather difficult to do this.[155]
- When in doubt, however, veer on the side of the system users rather than the system maintainers.

As to the mechanics of error handling we observe:

- Java and other languages offer programmers the ability to generate their own exceptions.
- Error processing should be as restricted as possible. Programmers should use a pre-written method rather than writing their own on an ad hoc basis.

One might solve the problem of messages arriving inappropriately by having a class method that simply reports the error – indicating the receiver, current state and message. One might then decide to shut the system down, depending on the likely consequences of this situation. The hope would be that this is an eventuality that would never occur.

One of the strengths of being able to throw your own exceptions is that exception handling takes you out of the code sequence in which you are engaged: it's a sort of respectable version of the "GoTo" command. The ability to switch directly to error handling in complicated situations can be worth diamonds.

---

[153] Imagine the possibilities for 'deadly embraces'.

[154] People commission systems that *work* rather than systems that are *robust,* and error handling costs money. On the other hand, most contracts stipulate post-delivery maintenance and support. Not handling errors now might cost you a lot of time and money later.

[155] The classic error message: 'A serious error has occurred' with the irritating, unspoken corollary 'and I am not going to tell you what to do about it!' is about as destructive as one can imagine. It's simply astonishing that software vendors get away with it!

© OUDCE, January, 2013

More on activity diagrams

The *Violet* tool does not currently permit the construction of activity diagrams. So far, we have used them once, in Chapter 4 (see Figure 4.4). We need to say a bit more about them.

Activity diagrams are useful for showing how actions are coordinated and they are often used to describe use case scenarios, which is how we used them in Chapter 4.

The following diagrams will provide you with a good test of your understanding of UML.

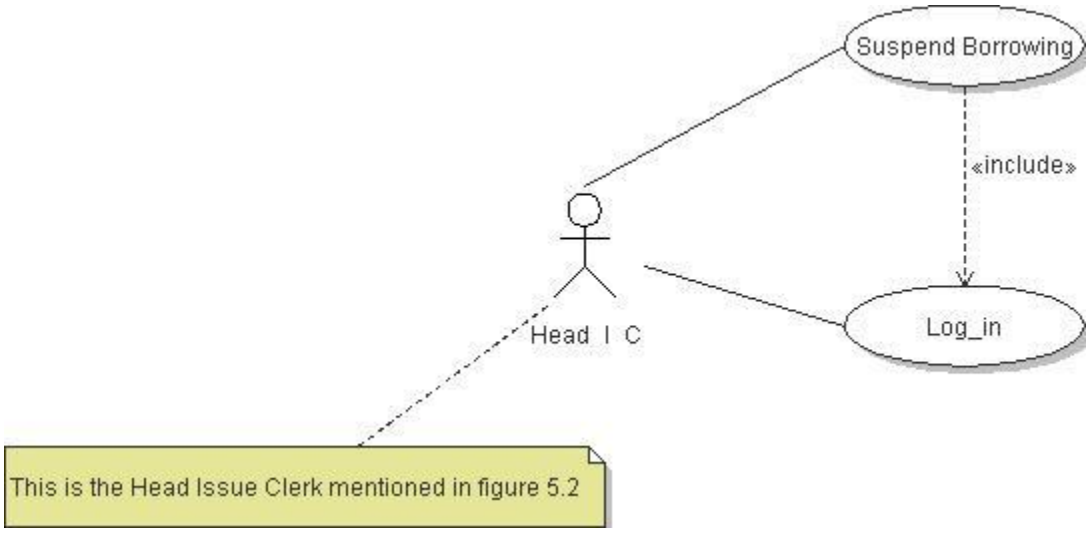


Figure 9.6: The Suspend Borrowing Use Case

Here is the corresponding activity diagram. We did it in *PowerPoint* and saved the slide to a gif file.

© OUDCE, January, 2013

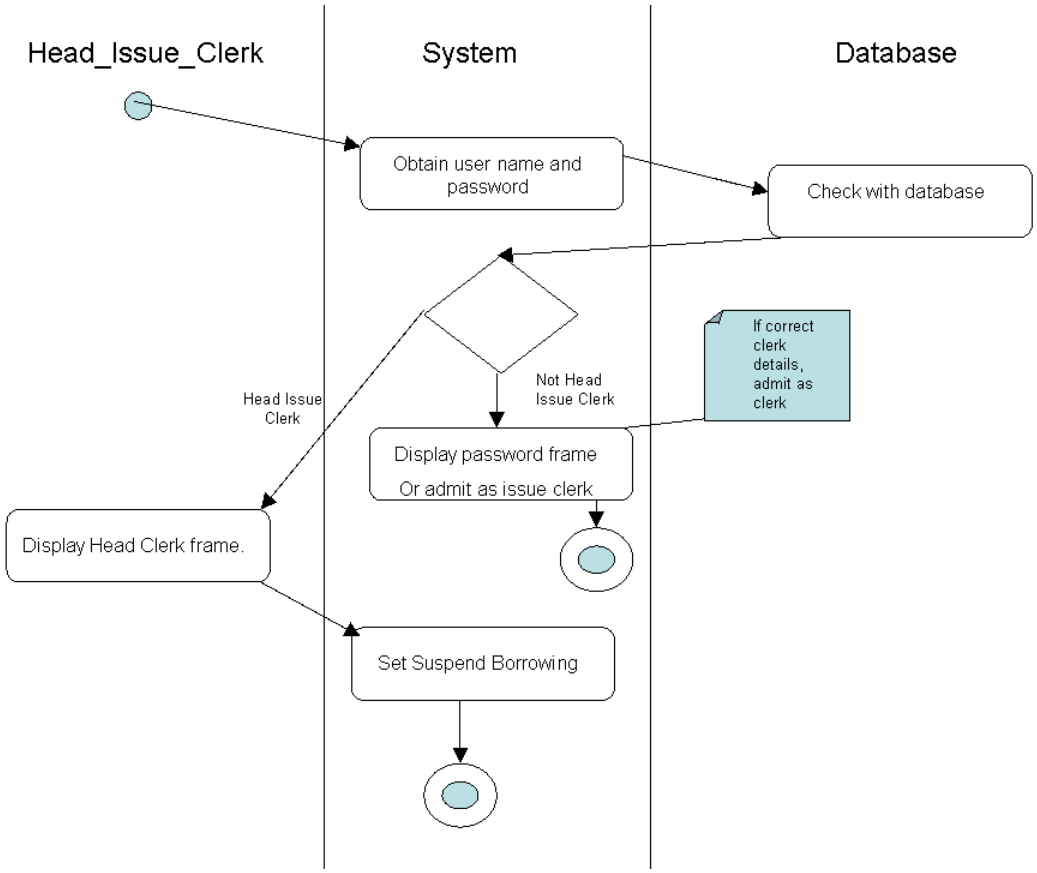


Figure 9.7: Suspend Borrowing Activity Diagram

Features to recognise in this activity diagram include:

1. the start of activity symbol (a single solid circle);
2. the activities symbol (a rectangle with round corners, with the activity name written inside);
3. decision points (diamond shapes);
4. the end of activities symbol (two circles, the interior one being solid).

It is supposed to be desirable to have a single start and a single stop symbol in your diagram, but in practice diagrams are frequently more intelligible when multiple stop symbols are used (as is the case here).

The annotation is supposed to tell you what happens when the supplied user name and password do not match those of a head issue clerk. Two possibilities exist:

- The supplied details match no one; in that case put the normal user name and password frame back onto the screen.
- The supplied details match an ordinary issue clerk; in that case put the standard issue clerk frame up onto the screen.

You might like to consider how Figure 9.7 might be amended by means of a further decision point in order to show these two possibilities more directly and remove the need for annotation.

© OUDCE, January, 2013

The implication of the activity diagram is that the Head Clerk frame includes a dialog box that permits the actor to suspend borrowing. (At this point, you have probably guessed what suspending borrowing entails.)

Forks and joins

Activity diagrams can have other useful features such as **synchronisation (forks and joins).** You may have activities that can proceed in parallel, and a point in time at which they must be completed. Figure 9.8 provides an illustration of how this is done.

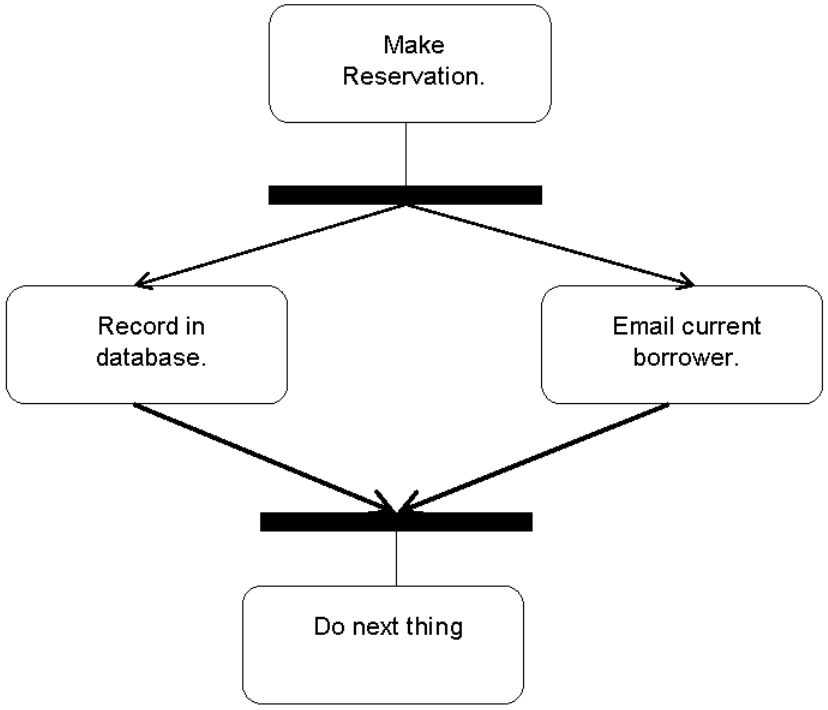


Figure 9.8:Activity Diagram Forks and Joins

The diagram is intended to express the idea that the **Make Reservation** activity forks into two subservient activities: Record in database (which records the reservation) and Email current borrower (which generates an e-mail to the person currently holding the book requesting its return.[156]) Only when *both* of these activities are completed do we continue with the processing.

If you really need forks and joins in your diagrams, by all means use them. They come up in parallel processing situations when several things are happening and things cannot proceed until all of them are finished. This occurs commonly in network computing but is rather rarer in simple applications.

[156] It would certainly not wait for the e-mail to be received!

Business work flows

Activity diagrams are sometimes used to describe the procedures relating to how people actually use the use of a system. Procedures of this type are often called **'business workflows'** and relate to high-level system usage, rather than lower level design. Time and motion experts and their modern, logistical, equivalents, are very interested in business workflows, but we have no space for them here.

The Suspend Borrowing use case

The Porterhouse Library specification talks about annual stocktaking. It might be useful to have a facility to suspend loans during that period and that is the use case that we have been considering. You already know that this activity would involve the Head Issue Clerk (referred to in the Porterhouse Library specification as the Head Librarian). You also know that we see this facility as being enabled by widgets available on a special display page which is only available to the Head Issue Clerk.

Here is the relevant sequence diagram.

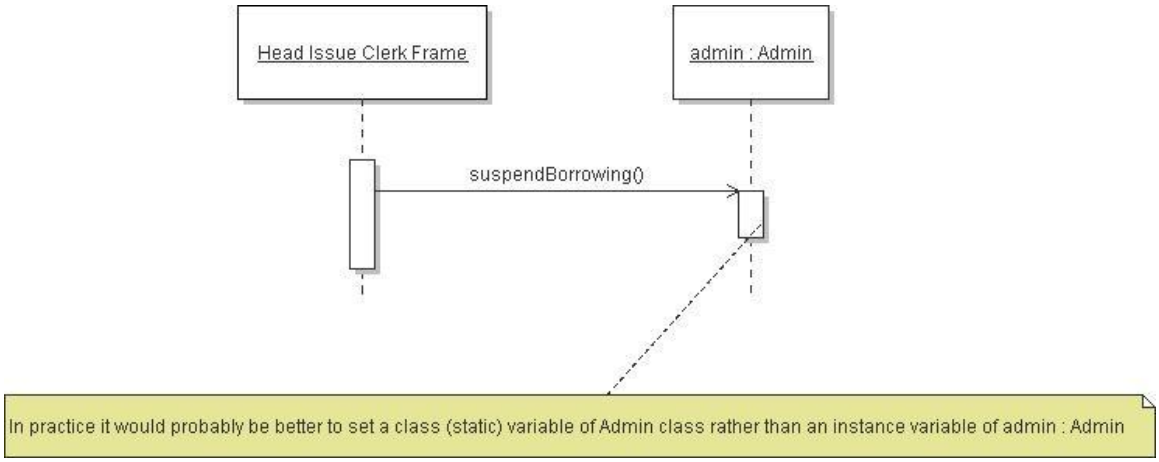


Figure 9.9: Sequence Diagram showing Suspend Borrowing

The Head Issue Clerk can suspend, and presumably reinstate, borrowing, by manipulating widgets in the password-protected Head Issue Clerk frame. We have shown the generated message as going from the Head Issue Clerk Frame to the single Admin class object, which we have called admin. In practice, the object corresponding to the Head Issue Clerk – headClerk : Head_Issue_Clerk – would probably have a listener method associated with the screen widget. When that was fired, the method would call another method which in turn contacted the suspendBorrowing() instance method of admin : Admin.

The supposition is that there is only one Admin object in the system, but it would probably be better for the suspendBorrowing() method to set a static (that is, class) attribute in Admin. Methods associated with loans would routinely consult with that attribute, and if borrowing were 'off' they would disallow it.

Here is the corresponding state diagram. Notice that we suppose that the system is initially in a state in which no loans may be made. The system state would perhaps change on receipt of a message from someone with the privileges of the Chief Librarian.

© OUDCE, January, 2013

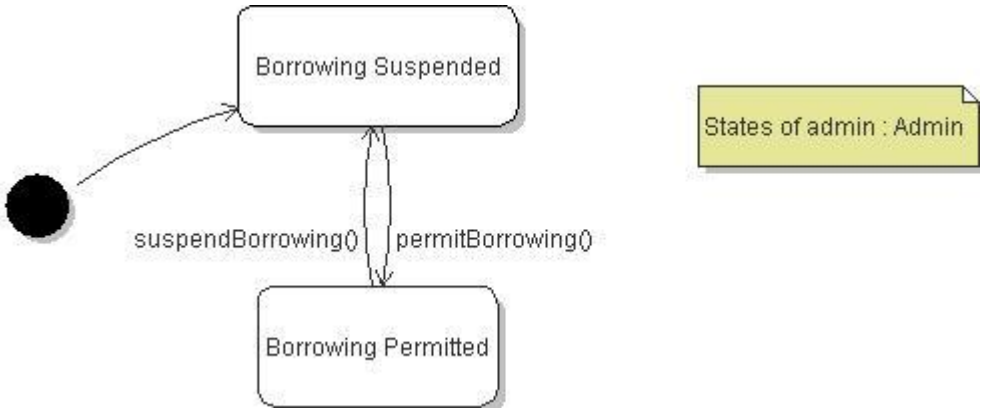


Figure 9.10: Some States of admin : Admin

One would probably want a library facility to make individual Volumes non-borrowable. This could be useful in reservations handling (since the specification speaks only of Volumes being removed from the shelves which is hardly a reliable system). It would also be useful when recording new acquisitions (since the specification seems to say that records concerning books are made even before they are physically present in the library).

You might like to consider how one would implement a non-borrowable facility that applies to individual books and whether such a facility is different in kind from the one just discussed (that is, does it correspond to making *all* Volumes non-borrowable?[157]).

Testing

Although they can occur elsewhere in the developmental life cycle, the UML diagrams that we have been describing are mostly artefacts of the analysis and design process.

At this point in our account, we find ourselves making the traditional obeisance to testing, while having left out any real coverage of it – which happens so often when people attempt to teach analysis and design.

Everyone agrees that testing should occur at *each* stage of the lifecycle, although that begs the question as to what is and what is not deemed to be testing.

Our claim is that most of what we have been discussing so far is pre-implementational. If that is really true, it's astonishing how many programming details have occurred in our account! If it is true, then the bulk of our testing will relate to validation – ensuring that the system we are building conforms to the requirements.[158] Our testing would then be **static testing** and quite innocent of code implementations.

On the other hand, our designs should engender a battery of **dynamic tests** associated with the expected behaviour of the delivered systems. If you look back at use case 1, you will see at once how functional testing is engendered by each scenario and you should be able to see how dynamic testing criteria would arise. Since our concern here is essentially pre-implementational, we shall say no more on this except that one would expect a process of test generation to be running alongside our analysis and design activities.

[157] In our opinion, this would be a silly way to do it!
[158] The distinction between verification and validation was touched on in Chapter 1.

© OUDCE, January, 2013

Validation

As an illustration of what validation involves, reconsider the use case we have just been looking at, which describes a system that does not permit loans to be made until the Chief Librarian authorises them. Is that what is wanted?

The testing regimes associated with object-oriented systems are similar to those of old-fashioned systems. Perhaps there is more emphasis on the early stages of system development but the essential details are the same. A key validation technique is the walkthrough and we have already mentioned walkthroughs in Chapter 8.

CARC cards, use case descriptions, activity diagrams, sequence diagrams and state diagrams can all be used to conduct walkthroughs for each scenario. Walkthroughs should be structured (divided into well-defined and identifiable steps) and planned so that participants understand their roles and are conversant with their support material.

Walkthroughs should be documented, and we recommend identifying one participant as a passive scribe. The scribe not only makes notes on the walkthrough as it occurs, but also writes it up later, perhaps analysing the problems that occur, resolving things that have not been decided, and reporting the results. The scribe would 'own' the walkthrough test data and be able to comment on it in relation to subsequent issues, such as regression testing.

One should aim to cover each use case and probably each scenario in this way. The reports associated with these walkthroughs are important items of system documentation. Walkthroughs that uncover difficulties that result in major design revisions must be revisited – together with regression testing of related scenarios[159].

The analysis and design techniques that we have been covering are based on the idea of use cases – that is, system functionality. One criticism of these ideas is that they compartmentalise the system too much and that this can produce inconsistencies and redundancies.

It is important to check that the overall system design is consistent and feasible, so at some stage a small 'magic circle' of designers needs to brainstorm[160] the completed design documents. We have explained how these documents might be produced, but managing and keeping track of them is just as important. Since the magic circle can hardly be expected to master all of the documentation, overview summaries need to be produced, with good references to the underlying documentation and with the odd sentient entity around to clarify things.

Like most assertions relating to the control of systems,[161] some of the clichés of testing theory rest on strongly held, and comparatively recent prejudices rather than scientific proof. Testing cannot establish the non-existence of errors, and the relationship between testing and cost efficiency is very variable and probably indeterminable.[162]

Perhaps we are writing better code than we did 30 years ago, but it does not feel like it! Of course, we have a responsibility to give you a view of 'modern best practice' with respect to testing because software contracts tend to expect that.

---

[159] You will remember that some aspects of testing were treated in the first unit of this course. You might like to revisit that material now.

[160] 'Blamestorming' occurs at the end of the project.

[161] And not just software systems!

[162] It is very hard to research what happens in testing in the industry because the data one needs is often regarded as too commercially sensitive – on the rare occasions where coherent forms of it exist. I actually did research I this for a few years, it was frustrating.

© OUDCE, January, 2013

It is clear to us that formal testing should be conducted by people without an emotional commitment to that which is being tested, and that illustrates the problems involved in coordinating activities pursued by humans – the central position that psychological effectiveness takes.[163]

The major deficiency in what we have been telling you probably lies in requirements and in testing. If you consider studying this subject in any detail, the next topics that you should consider include requirements analysis, and a more systematic approach to specification and testing.

Specification of methods

Our analysis of the library system has already zoomed system functionality down to the level of individual methods. You can see an example of this in Figure 6.2, where four interface methods are mentioned. Our analysis and design documentation should have to include fairly precise specifications of these methods.

The techniques used to produce such specifications are fairly standard. At the most basic level one needs to discuss a goods-and-services view of the methods – including information about what they require and what they supply.

There are well-established techniques for doing this and you have probably met them before. One would probably:

- specify the *pre-conditions* expected before the method is called;
- specify the *post-conditions* resulting from the method running;
- indicate *what* the method is supposed to do;
- indicate *how* it is going to do it, probably using some sort of notation or pseudocode;
- specify state changes relating to the method;
- specify such other methods as it would use;

The correctness[164] of the resulting code will be tested during the **verification** process. Modern languages, such as C++ and Java, have facilities for embedding logical assertions regarding pre-, post- and mid-term conditions within the sequence of the code in order to structure its logical development. These conditions can be very helpful during verification and can be disabled when the code is finally delivered, but using these ideas is something of a skilled activity and programmers able to handle them are few and far between, and expensive.

Our view is that these ideas can be useful but that some part of their effectiveness may be a direct consequence of the simple act of spending time thinking about code in a formal way, rather than just writing it. In that sense, *any* technique that formalises spending time on code writing is likely to be as good.

Code specification is an important part of analysis and design but the techniques used are not unique to object-oriented systems so we shall say no more about them here.

System documentation

We have spent nine chapters discussing techniques associated with modelling systems in object-oriented terms. Along the way we have said quite a bit about the basic UML diagrams and the sort of facilities offered by programming languages like Java. Our account has been based on use case modelling, but there are other ways to model systems and you would encounter different prioritisations and ideas in any real working environment. One can make a student

[163] We have personal experience of designers not being informed of errors due to personality clashes, and many big organisations inculcate a culture of 'not rocking the boat' as the financial crisis of 2009 amply demonstrates.
[164] Does it behave as specified?

exercise impossibly neat, but real-life analysis and design can be a messy, exhausting and incoherent, process – and one in which you lose friends!

We have not said nearly enough about the analysis and design process itself and how it is managed. There simply isn't time for a proper treatment here and we prefer to concentrate on models – hoping to return to managerial concerns in other courses.

If you were to attempt the construction of a system using these ideas, you would find yourself producing a considerable number of related documents, which must be managed, cross-referenced, accessible and clear. More than anything else, documents should be 'owned', so that responsibility, authority and consistency are established.

If you were to attempt such an exercise in a team, you would be well advised to establish team responsibilities at an early stage, and make strenuous efforts to **re-establish** them **periodically** as your project ages.

Cross-referencing is a key issue. Web hyperlinking can make this far easier to achieve effectively than was the case in the past – and electronic documents are easier to search than paper ones.[165] You should also construct some kind of simple index of what is in the system documentation, and where it is (and possibly, who can see it). It's very common for project team members simply to be unaware of the existence of important material. It is very common for one part of a team to be working away at something that the other part has long since rejected.

Summary

We have now finished what we have to tell you about analysis and design and about UML. In our final chapter we shall introduce you to distributed computing and tell you something about the exciting developments and opportunities offered by the modern Internet.

You should supplement your understanding of UML by reading some of the tutorials available on the Internet: a simple search should uncover tens of them. Do try to restrict the techniques you use in group work to those that we have described because then we can be sure that everyone will be able to understand the diagrams you produce.

[165] But note again that documents must be **accessible**.

© OUDCE, January, 2013

Chapter 10: Distributed Computing

A distributed system is usually taken to involve a collection of independent computers offering services that appear to the recipient to emanate from a single computer. Such systems will be our prime area of consideration here, although what we have to say applies in more conventional networks too.

We shall try to outline some of the key ideas of modern distributed computing. You will realise that this is a considerable task and that we shall have to be brisk and perhaps even occasionally economical with the truth. Many modern business applications are distributed and use objects. It is important that you obtain some understanding of what this involves; although you must be aware that the situation is fast-changing.

Enterprise frameworks

An Enterprise framework is a collection of software facilities that permits programmers to write software involving distributed programs.

Most programming in distributed systems involves object-oriented ideas. This is partly due to the features that objects offer – such things as: security, modularity and encapsulation – and it is partly due to the fact that distributed systems have taken off in the period when the bulk of major programming languages offered support for objects.

Object-oriented languages, in turn, offer a lot of pre-written support for distributed computing. For example, Enterprise Java 2 (J2EE) technologies have a range of facilities catering for much of what one wants to do.

These include:

- naming and directory services (locating services in a distributed environment)
- security services (validating data and transactions and keeping them confidential)
- database access and programming
- mail handling (sending and receiving e-mails from programs)
- XML processing (data transfer and document handling)
- component-based applications (Enterprise Java Beans)
- remote message sending (invoking a method on an object on another computer)
- dynamic web services (applets, servlets and Java Server pages)

The Microsoft Corporation's .Net framework offers a similar range of capabilities. Although .Net services are perhaps more seamlessly integrated with each other, .Net suffers in being, essentially, a pure Microsoft product and being seen to have come rather late to the party. Java Enterprise has the support of a wide range of influential software companies and more flexibility, so we shall not discuss .Net further here restricting our attention mainly to Java products.

Since the first version of this document was produced there has been a significant increase in the tendency of people to resolve software engineering problems with web-based technologies. There has also been a corresponding tendency to vest application logic in modern, web-oriented, lighter, technologies such as PHP, Python, Ruby or even Perl, rather than using the heavyweight utility of Java.

Nothing that we have said is invalidated by this, but had we written the unit more recently, we might well have chosen to illustrate some of what we say using these modern languages; and you should be aware of this trend in contemporary software engineering.

© OUDCE, January, 2013

LANs and WANs

Distributed computing is network computing. Networks are configurations of **stations** linked by various kinds of communication media, such as wire, fibre optics or radio.

Network communication is always **serial**: bits move sequentially.[166] Most networks have bounds on the size of the information that may be sent at one go so most network systems need to package data into a series of chunks, variously called: packets, datagrams or segments.[167] For our purposes, there are two sorts of networks:

10.1 Local Area Networks (LANs)

These *often*[168] involve:

a. a small geographical area, typically less than one square kilometre
b. networks under the control of a single manager and located within a secure environment
c. networks involving the same computer systems and software
d. networks involving a single communication medium
e. networks in which the route between two stations is well defined and probably unique
f. networks involving quite a range of possible proprietary and open source software
g. network protocols in which individual stations must wait for some sort of clearance to transmit
h. network protocols which **broadcast** to all stations – with stations actually reading messages only when they are addressed specifically to them
i. data transfer based on **timing** considerations operating at speeds of between 10 and 100 million bits per second.

10.2 Wide Area Networks (WANs)

These *often* involve:

a. a large geographical area, perhaps spanning continents
b. networks under the control of distinct economic concerns
c. networks involving a variety of computer systems and software
d. networks involving a variety of communications media
e. networks in which the route between two stations can vary, even for packets of data forming parts of the same message
f. networks using the TCP/IP protocol suite
g. network protocols in which stations appear to transmit whenever they wish.[169]
h. network protocols which operate in a point-to-point mode between stations, so that a particular message is sent and delivered to a particular station
i. data transfer based on **routing** considerations involving intermediate stations and public data highways, with speeds varying from about 64 Kbps upwards

The modern Internet is a wide area network. More specifically, it is the largest example of *an internet* – a connected network of networks.

---

[166] Parallel transfer is restricted to extremely short distances due to synchronisation difficulties: one can attempt such transfer from a CPU to a printer, along a short cable, but no further.
[167] It is not appropriate to get fussier about the terminology used in this context than the available literature. Were this a network course, we would need to be more precise, and these terms might turn out to represent rather different ideas.
[168] These points are really just generalisations and are more or less true in particular cases.
[169] The first step in a WAN is usually a LAN, so LAN protocols might intercede here to restrict transmission until clearance, but the *network* level programming is essentially spontaneous.

© OUDCE, January, 2013

The key idea of internetworking is that the constituent networks run their own, local network systems and communicate with stations on other networks by means of higher level (**network layer**) control information, and the services of intelligent intermediary stations known as **routers**.

Messages

Applications communicate over networks in a number of ways, of which probably the most important is **message-passing**. Message-passing usually involves a fixed communication language which is understood at each side of a link, that is, a **protocol**.

In one sense, any communication involves messages; but the messages we are discussing are essential textual communications with pre-specified structure.

Messages can be either **synchronous**, in which case the sender waits for a reply before proceeding, or **asynchronous**, in which case the sender goes on to do something else and receives a reply later, when it becomes available.

Object-oriented message-passing is done in a synchronous way. Asynchronous messages occur, for example, when you use the **print** command, (your computer does not immediately lock until the printing operation is physically finished).[170]

A telephone message would be synchronous; a postcard would be asynchronous.

Message passing fits in well with the central **client-server** view of distributed facilities, in which one station (the client) solicits services from another station (the server). When you use your browser to request a web page, your browser is in the role of a client sending a HTTP message to obtain the page.

Message-passing is an immensely simple way to communicate but it has some drawbacks; among them that both sides of the link must speak the same protocol language – so alterations and additions are very difficult, bearing in mind that the web context would imply stations that belonged to disparate individuals or organisations. One can illustrate this by suggesting a change to the popular hypertext transfer protocol (HTTP) mentioned above: one would have to re-program every web server and, even worse, every browser.[171]

Protocols

Theoretical and practical considerations have resulted in a broad general consensus as to the facilities one wants from electronic communication and the best way to organise them. It has generally been agreed that the best practice is to divide communications options into areas of similar responsibility, called **protocol layers**. Each layer has its own control information that is added to the transmitted data as a **header**. This can be thought of as a similar process to the one undergone by letters in the course of delivery, when they may be placed in envelopes, post-boxes, mailbags, delivery vans, etc.

There is a hierarchy of protocol layers. Lower layers deal with the basics of communication – such things as relating electrical signals to bit settings and the length of time that such signals need to be maintained on a wire. Higher layers relate to more sophisticated delivery options – such things as guaranteeing data integrity, sequencing and delivery.

[170] Well, it shouldn't!

[171] Web protocols often contain control information identifying which version of the protocol they represent and this helps the re-programming problem a little bit because stations can reject protocol versions with which they are not equipped to deal. Whether they have actually been programmed to do this is an entirely different matter!

© OUDCE, January, 2013

The information within protocol headers encodes data relating to services. It is constructed by software in the sending computer and directed to software lying in the same conceptual layer at the receiving computer, but on the other side of a communication link. So, header data is added at the message source and removed and processed at the message destination; and it forms a control message that must be added to the actual message. Protocols are really communication languages.

Data sent across the Internet might look like this:

Figure 10.1: Data Showing Protocol Headers

Any trip across the Internet consists of a sequence of sub-trips such as those:

- from the sending station to a **router**, probably across a LAN
- from the router to another router, possibly across an Internet backbone link[172]
- from a destination router to the receiving station, probably across a LAN

The data link header and trailer would be added to the data in order to handle the smallest of these sub-trips – quite possibly traversing a LAN. Typically, this data might refer to the **Ethernet** family of protocols. It would then include synchronisation information, check information and local network addresses (**MAC** addresses).[173]

The network header relates to rather longer journeys. This is the information that permits internetworking, so it must contain the information needed for a journey across the Internet, including original and destination **IP addresses** and (probably) information about header integrity.[174] The network header contains information that routers read and update[175] and each such update corresponds to an Internet **hop**.

A trip across the modern Internet usually involves fewer than 30 hops, but may take considerably more.[176]

The transport header information relates to the transport protocol needed, and that in turn relates to the transport services that are required.

[172] And this situation is likely to occur several times during the transmission of a single packet of information.
[173] Media Access Control – essentially, LAN addressing.
[174] This is not done in version 6 of the IP protocol, but most people are still using version 4, where it is done.
[175] So the network header is read and processed by *every* intermediate router in the Internet. These decrement hop counts and recalculate header checksums – quite a processing load.
[176] The upper limit for both IPv4 and IPv6 is 255 hops.

© OUDCE, January, 2013

If data integrity is essential, the transport protocol used will probably be **TCP** and the header will then include data integrity[177] and size information.

The transport header also identifies the application to which the data packet is handed on arrival. This is done by means of **port numbers**.[178] The transport header is set at the sending station and unpacked and processed by the receiving station. So transport protocols involve dialogs between the stations at either end of the transmission link and are of no concern to intermediate stations, such as routers. Intermediate stations see transport data as simply part of the data load of the message.

The hypertext transfer protocol usually runs on **port 80**, and if that is the setting, then the software counterpart of the TCP protocol passes the data it receives to the web server or browser program at the message destination (depending on whether it is a request or its reply).

Web page requests involve an **application protocol** called the **hypertext transfer protocol (HTTP)**. This protocol involves a further header[179], so if this is a web request one would expect the blue data field in figure 10.1 to be further subdivided into a header and additional data (which would, perhaps, be a web page for display). We have not shown this.

A request for a web page is made by a browser[180]. That would formulate the request using HTTP and pass it to its local transport layer programs. They would establish a TCP connection with the destination server (where the page is to be found) and package the HTTP request along with a transport header. That data would then be handed to the local network layer programs which would add their own network layer header and pass the data on to data link programs which would also add their header, and so on.

At the receiving station the incoming data is processed in a reverse way. Data bubbles up through the software in the different layers in the order: physical, data link, network, transport, application (HTTP – that is, a web server). Software in each layer removes that layer's header, processes it, and hands what is left to software in the next layer up. The TCP program finally reads the port setting as 80 and thus knows to hand the data (an HTTP request) to the local web server.

You will notice that this process of adding and removing headers in some sense follows a **first in last out** regime, that is, it resembles the data structure known as a **stack**, which is why people talk of **protocol stacks**. Protocol stacks indicate what one *can* do, and where and when to do it. Here is the protocol stack often associated with Internet communications. We offer an example of a protocol within each layer of the stack.

---

[177] The IP header may contain information about *its own* integrity, but, in contrast, the TCP header contains information about the integrity of *the data* itself.

[178] Numbers identifying applications. There are a large number of predefined identifications, and additional numbers that can be used as you wish. Port 80 is famously used to indicate Web processing. A web search on 'well-known ports' will give you more information about this, if you feel you need it.

[179] And is an **application protocol**, logically above the transport protocol in the *protocol stack*.

[180] The maxim: *cherchez les actors*, applies here. Browsers are usually operated by sentient life forms but of course it is possible for software to generate an HTTP request quite independent of a browser.

© OUDCE, January, 2013

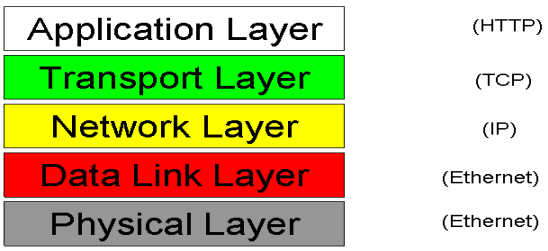


Figure 10.2

The protocol layers shown involve the key layers associated with Internet communications and so provide a model of how information is moved around on the Internet. There are other protocol models of data communication, including an influential seven-layer model which, essentially, replaces the upper part of the transport layer and the bottom part of the application layers shown above with another two layers – the **session** layer and the **presentation** layer. These ideas do crop up in Internet communications[181] but are not generally separated from Transport and Application functions.

Hypertext Transfer Protocol (HTTP)

HTTP is the great workhorse of Internet application programming. Its original use was restricted to the manipulation of web pages – marked-up documents. Now it is employed in all aspects of Internet business and one can achieve a clear view of how distributed computing works if one starts with a clear view of HTTP.

HTTP has some features that are true of most Internet protocols.

1. It involves headers but not trailers.
2. It is expressed in simple ASCII text and is readable.
3. It involves a simple client-server model – a single request gets a single reply.

Point 3 is particularly important. HTTP is a **connectionless** protocol. HTTP requests are **stateless**. If you contact the server again, it will have no memory of you! This is a highly important feature of web transaction processing.

HTTP was originally designed as a simple document handling protocol. The options are case sensitive. The following table of HTTP request options is a slightly modified version of one taken from Tanenbaum (5).

HTTP Request	Original Meaning
GET	Request to read a web page.
PUT	Request to write (store) a web page.
HEAD	Request to read the control information in a web page.
POST	Request to append data to a web page.
DELETE	Request to remove a web page.

[181] Indeed they are particularly relevant to what we are discussing because Enterprise computing really does need to differentiate **session**, **presentation** and **application** processing.

© OUDCE, January, 2013

The commonest HTTP request is for the retrieval of a page. The associated HTTP header *might* then look something like this.

GET /www.tall.ox.ac.uk HTTP/1.0

There are actually *two* lines in this header – the textual one you can read **and a blank line**[182].

This particular request refers to the simplest form of HTTP, version 1.0; which is now virtually obsolete.[183] Most real GET requests would involve HTTP/1.1 and therefore several more lines of information relating to the characteristics of the browser and the information that can be sent back. This request is for the root web page of the server www.tall.ox.ac.uk.

Successful attempts to obtain pages can expect to see replies that start:

HTTP/1.1 200 OK

which explains that the server is using version 1.1 of the HTTP protocol and that the request was successful. The reply header would go on to contain several more lines relating to the characteristics of the server, the time and the size and nature of the data being returned.[184]

The reply header ends in a blank line, which is followed by the page data itself. The complete set of data might involve several packets of information being sent from the server to the browser. These are checked and ordered correctly by the TCP processing on the client side.

HTTP headers can additionally contain small bits of additional data that are stored and retrieved from nominated areas of the client's hard disk and which are often used to maintain state information; these small bits of data are called **cookies** and we shall return to them later.

Two-tiered architectures

The original view of HTTP involves a simple series of requests and their resolution. Little or no data processing occurs in such a pure information retrieval system. It is this **two-tiered** architecture – browser and server, which characterised early web business applications[185].

Anything more complicated must involve *processing*, either at the client side or at the server side, or both. Processing involves information retrieval and storage, and state information – so it involves **application logic** and therefore **application processing**.

Client-side processing

As the web began to take off, software companies began to find ways to extend its capabilities beyond simple hyperlinking and document retrieval. One of the first ways in which this was done involved embedding small programs in web pages, which were executed at the client end under the auspices of the browser. Programs of this sort could add vitality and dynamism to web displays or provide simple, local, calculating and display functions.

The first such programs were written in a language that came to be called JavaScript.[186] Later on people developed technology to embed small Java programs, known as **applets**, within web pages.

---

[182] Indicated by control characters that a careless viewer might miss.
[183] We use it here simply to keep things simple.
[184] Using a coded classification called MIME, probably text/html, in this particular case.
[185] The web was used to showcase products that were then purchased by post or by telephone sales.

© OUDCE, January, 2013

Java was appropriate for client-side programs due to its portability and the fact that its good security features mean that browsers can be fairly sure that Java applets cannot do any malicious damage. These same security features do limit what one can do with applets, however, and there have been attempts to permit applets to have more scope, on the basis of permissions granted by various sorts of security certificate.

Microsoft developed similar client-side programming facilities based on their proprietary ActiveX controls. These were certainly regarded as less safe than applets when they first appeared.

Another way in which the capabilities of web pages were extended involves a clever exploitation of HTTP, using **HTML forms**.

HTML forms allow the transmission of information from the client browser to the server. Data is entered into a web page and sent to programs running at the server.

Forms offer the basic WIMP display facilities that we covered in Chapter 6 (textboxes, radio buttons, etc.), and include information about how to package the data and where to send it.

One could produce a web page that closely resembles the interface frame in Figure 6.1 of Chapter 6. This would have been useful had we adopted multiple access Option 8.3 described in Chapter 8.

Here is our attempt to do that; no doubt you would be able to mimic the frame in Figure 6.1 more precisely. The page is: login.html and you should be able to display it in your browser and enter data into the fields. Don't expect it to work as a password processor! It is directed to a **servlet program** that does not exist!

[186] The name has more to do with following a trend than any real relationship to the Java programming language.

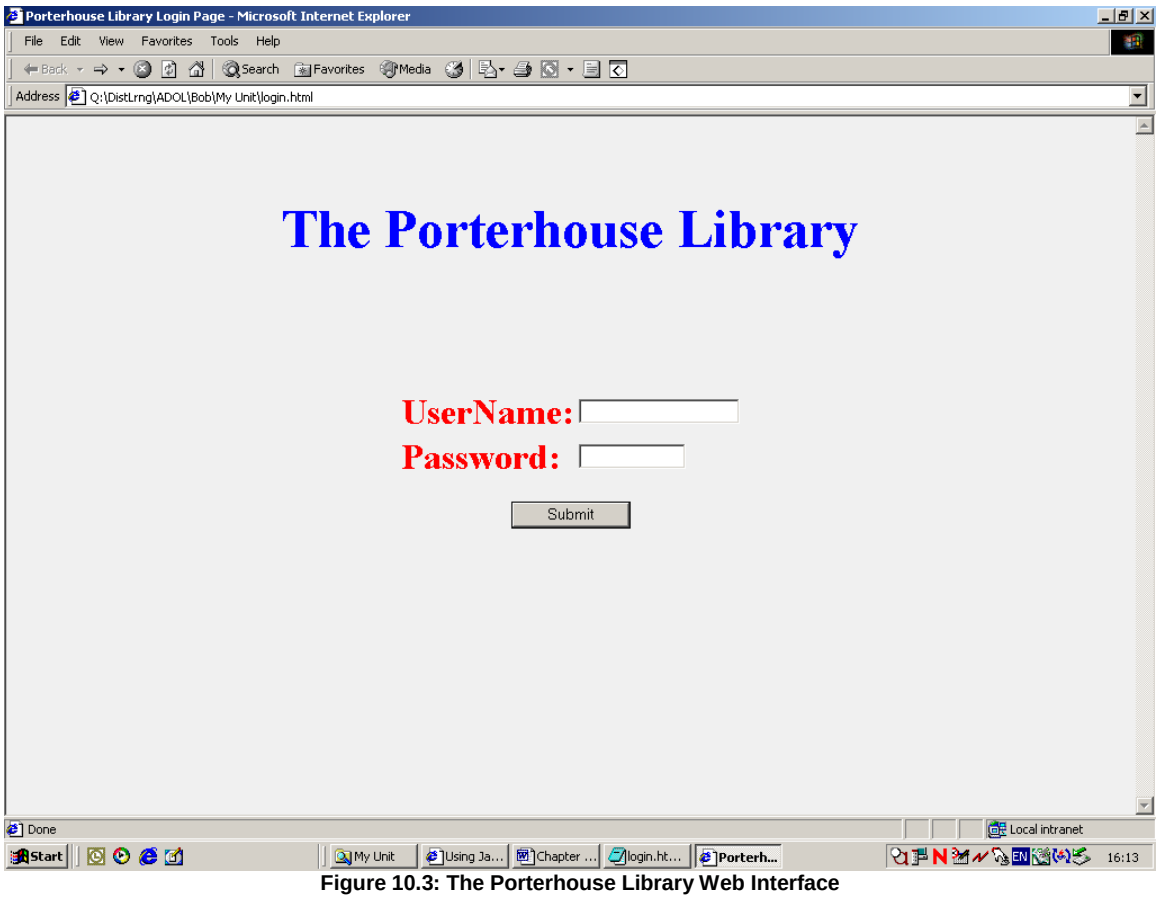


Figure 10.3: The Porterhouse Library Web Interface

Form data is sent by using *either* an HTTP GET request (in which case the data masquerades as an Internet address and the request appears as if it were a request for a web page) *or* an HTTP POST request (in which case the data is packaged as if it were an addition to an existing web page).

The GET method was developed first. It is regarded as insecure because it makes form data clearly readable. You should not adduce, however, as some books seem to suggest, that the technique is obsolete. It has clear advantages for non-sensitive, static data, and GET requests can be bookmarked.[187] GET requests involve data dressed up to look like URLs. There are practical limits as to how much data one can send in this way – at best, a couple of thousand bytes.[188]

The POST method is regarded as slightly more secure, and it is generally appropriate when large amounts of data are to be sent, or when the submitted data results in something other than the return of static information (for example, web shopping).

When security is a serious issue, as it would be for such things as password authentication or credit card processing, one should use additional encryption techniques rather than committing card numbers to web forms directly. These techniques often involve moving to a connection operating using the **Secure Sockets Layer (SSL) protocol** – a security protocol based on **public key cryptography** and sometimes involve the offices of trusted third-parties

[187] You can bookmark the results of searches and re-apply the search by simply resuscitating the bookmark. This sends the original search parameters back to the search engine.
[188] To set this in perspective, a typical page in the *Word* version of this document would involve around 3,000 bytes.

© OUDCE, January, 2013

The crucial point about Enterprise usage of the Internet is that not much processing occurs on the client side. The small amount of client processing in typical web transactions would involve forms, JavaScript (to pre-process data before transmission is attempted) and perhaps some security encryption. HTTP-based transactions are said to involve **thin clients** because the bulk of the processing must occur at the other side of the link – with the server.[189]

Server-side processing

The first initiatives in web processing involved the **Common Gateway Interface** (CGI). This is essentially the processing we outlined above, involving forms. Servers, seeing CGI data, would strip out the identity of the program for which it was intended, start that program up, and present it with the information that had been submitted.

You will recall that form data is submitted using either a GET or a POST request, both of which involve replies. Typically, replies would result in the production of a web page that would be sent to the client browser and displayed by it.

We could utilise the services we have described to implement an application that reports on the closing prices for commodities in the New York Stock Exchange. The CGI program would simply read the relevant information from a database and write it to the page that is sent back to the client.

This sort of application involves a **three-tiered architecture**:

1. A *thin* client, which communicates using HTTP and does little more than present data (the **presentation layer**).
2. A server, also using HTTP, with programming capabilities (the **application server** layer).
3. A repository of static information – one or more databases (the **database** layer).

This three-tiered architecture is the basis of modern web transactions. People can, and do, complicate the story, but any processing system you are likely to see will probably have these features.

When people discuss more complicated variants of this architecture, they do so by dissecting the activities of the second layer. Database access introduces a delay into processing; it is probably desirable for the process associated with this to be away from the server processing.

If you are familiar with the seven-layer model of data communications that we mentioned in passing, you will recognise that the application server layer needs to take on responsibilities associated with both the presentation layer (formatting replies as web pages) and the session layer (maintaining state across the transaction).

You might also agree, when you have read what follows, that it is fairly hard to view Internet communication in this seven-layer stack way. One consequence of thin client processing is that presentation and session processing tend to be unfairly weighted on the server side of the link and so it is correspondingly hard to view these issues in communication terms.

[189] The possibilities for downloading code are small and we can hardly expect customers to do it for our convenience.

© OUDCE, January, 2013

State maintenance

The stateless nature of HTTP makes commercial data processing much trickier than it might have been. You will have encountered **shopping carts** in your own forays into the Web. These involve a series of HTTP transactions, probably involving forms, but how are these separate transactions identified as related?

The short answer is that a number of 'tricks' are used! These include:

- Perpetual re-identification, which is a requirement that the user re-enter a session-tracking identifier every time the user returns an HTTP request. This is tedious, and now not much used.
- Cookies, which are small storage areas allocated for the use of the server by the client and subject to size and date rules. Cookies are automatically entered into HTTP headers by both clients and servers.
- Hidden fields, which are pages sent back from the server to the client which include data that is not displayed on the screen but which is then unconsciously returned by the client in subsequent transactions. This evolved from the idea of re-identification, but it is messy and likely to cause data transfers to be considerably larger than they need be.
- URL rewriting, which is the dynamic generation of URLs at the server in order to reference a 'session' or even encode previous user data.

Java Enterprise facilities include session-tracking abilities in a way that is almost automatic. It uses some of these techniques.

Our model for Enterprise applications looks like this.

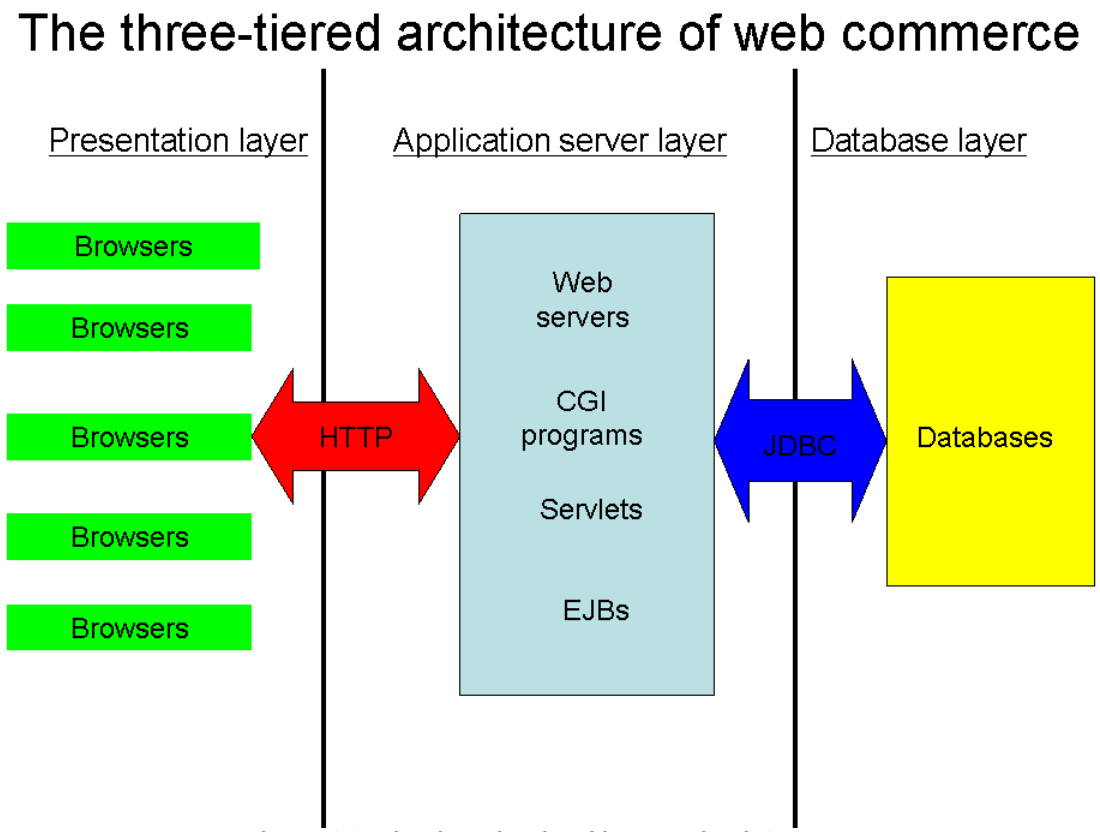


Figure 10.3: Tthe Three-tiered Architecture of Web Commerce

© OUDCE, January, 2013

In this chapter, our programming language is Java, so the connection between the application server layer and the databases uses **Java Database Connectivity (JDBC) –** Java facilities to work with relational data. You saw an example of what this involves in Chapter 8.

The connection between the presentation and application server layers uses HTTP. The facilities listed in the middle layer are often facilities from the J2EE framework.

We shall now give an outline of these technologies.

Servers

Although there are many commercial web servers, such as Microsoft's *Internet Information Server (IIS)*, the world's most popular web server is available at no charge at all as part of the open source movement. It is the famous *Apache* web server.

It may be that you have never thought of downloading and installing your own web server and imagine that it would be a tedious and fraught business. On the contrary, *Apache* is extremely easy to run and install and does not take up much space. Once you have installed *Apache*, you will be able to try out some aspects of web programming on a single machine.[190]

Apache is a web server, but some of the Java tools shown in Figure 10.3 require other sorts of server in order to function. For example, servlets are served by a servlet engine such as *Apache Tomcat*. This open source product is also easily obtained as a web download. It has basic web serving ability built into it, and it coexists with the *Apache* web server very well[191].

CGI programs

We discussed CGI programs earlier. Forms direct their data to these programs using URLs. Data is sent to the web server, which starts up the required program. Servers usually keep CGI programs in a special folder, usually called something like **cgi-bin**. CGI programming[192]involves writing the program, placing it in the required folder and constructing some web pages with forms that reference it.

CGI programs can be written in any language that is supported by the server. Languages like Perl and PHP, which have good text-handling capabilities, are particularly popular; but CGI programming is inherently inefficient because programs have to be loaded and unloaded on demand.

A popular server would find itself dealing with many copies of the same program, all doing essentially the same thing. Efficiency dictates that we would want a set-up with exactly one copy of the program that remains permanently in memory rather than being continually loaded and unloaded,[193] and that the single copy of the program should be utilised in a **multi-threaded** way.

The Java programming language has built-in support for multiple concurrent uses of individual programs – **threads**. We did say something about threads in Chapter 2, and you might like to re-read that now.

---

[190] One of these tricks involves your client browser talking to your server through **loopback** addresses – so data never actually leaves your machine at all. You can do most web programming entirely on a single machine, if you so choose.

[191] So well, that many people regard *Tomcat* as a sort of *Apache* plug-in.

[192] Do not confuse this term confused with Computer Generated Imaging – the CGI used in the recent spate of fantasy movies.

[193] Loading programs is an expensive business in CPU terms, particularly when you seem to be doing little else!

© OUDCE, January, 2013

Several different and concurrent programs start when *any* Java application runs. For example, Java garbage collection runs continuously as a background task. But Java also gives programmers the ability to permit several threads of execution of any given code.[194] They do this *either* by using inheritance or by the implementation of a special thread interface.[195]

Execution threads make some memory and space demands on the operating system – but far fewer than those associated with running an individual program.

There are techniques and disciplines associated with this sort of programming. For example, one has to be careful about data items; one could, potentially, set them to particular values, only to find that another thread chipped in, changing those values. When your execution sequence resumed, you would to find data that seemed to have been miraculously altered. We shall not go into the details here other than to say that Java does offer ways to safeguard against this problem.[196]

Thread programming is particularly useful for web servers. A popular server receives thousands of requests a day, and the Internet communication model that we have described suggests that it gives its attention to each one, setting up the TCP connection and finally sending the requested data before going on to the next. Demand levels are such that this would be unworkable. What actually happens is that the server recurrently 'listens' for requests on the web port 80. When a request comes in, the server spawns a thread to deal with it, using another port. It then goes back to listening on the original server port. At a given instant the server might be dealing with many concurrent requests, each one of them using a distinct port, and each processed by a distinct thread. Threads terminate at request end, of course.

Servlets

Servlets can be thought of as the server-side counterparts of applets. Like applets, they do not themselves have **main** methods, and they rely on the good offices of other programs to run.

Applets run in browsers; servlets run in an environment provided by the servlet engine. Servlets are inherently multi-threaded.[197]

Servlets are small Java programs that run continuously[198] at the server. Form data may be directed at particular servlets. When the browser receives a servlet, it knows to direct it to the servlet engine, which in turn sends it to the appropriate servlet program.

Option 8.3 of Chapter 8 would require a web page that remote users could use for logins. The option speaks of 'remote access via the Internet'. One surmises that Customers would be restricted to activities associated with data interrogation and book reservation, although it is just possible to imagine that some classes of Customer could extend loans, or that Customers could pay fines remotely.

We presented you with a web login page in Figure 10.3. This was consciously modelled on the interface frame we showed you in Figure 6.1. It has forms relating to user name and password.

Not having much expertise with editing programs such as *FrontPage* or *DreamWeaver*, we constructed the page in Figure 10.3 by hand, so don't imagine that we regard the mark-up as in

[194] If you have a Microsoft system, you might like to use the Task Manager (from the Control Panel) to check on the threads running in your own system. You will find far more threads than processes! Note, however, that these are not Java threads.
[195] The lack of multiple inheritance in Java demands a thread technique that is not based on inheritance and, as usual, Java interfaces do the job.
[196] If you want to look this up for yourself, lookup the Java keyword **synchronized**.
[197] Single-threaded servlets are possible.
[198] They do not usually need to be loaded when needed as they are already there!

© OUDCE, January, 2013

any way exemplary! We are simply illustrating how things could work rather than attempting to show best practice.

Here is the HTML for that page.

```
<html>
<!-- Demonstration web interface page. Please note - this does not demonstrate HTML excellence! The font tag is now deprecated, for instance!-->
<head>
<title>Porterhouse Library Login Page</title>
<script language = "JavaScript">
<!--
function validate() //a simple JavaScript validator
{
  alert("JavaScript is validating the content of your form.");

  if (theForm.username.value == "")
   {
     alert("Your User Name field is incorrect!");
     return false;
   }
   else
   {
     if (theForm.password.value == "")
        {
          alert("Your Password is missing!");
          return false;
        }
     else
     {
        alert("Your form looks correct, but there is No SERVLET - So Submission Fails!");
        return false;
     }
  }
}
//-->
</script>
</head>
<body bgcolor="F0F0F0">
<br>
```

```
<br>
<br>
<center>
<h1> <font color= "blue" size="11" >The Porterhouse Library
</font></h1>
<br>
<br>
<br>
<br>
<br>
<form                                                    id="myForm"
action="http://www.conted.ox.ac.uk/servlet/Porterhousepwservlet"
method="POST" name="theForm" onsubmit="return validate(this);" >
<table> <!-- Use a table to attempt to line things up properly! -->
<tr>
        <td><font color="red" size="6"><b>
                              UserName: </b></font></td>
        <td><input type = "text" name="username" ></td>
</tr>
<tr>
        <td><font color="red" size="6"><b>Password:</b></font></td>
        <td>
        <input type = "password" name="password" size="12"></td>
</tr>
</table>
<br>
<input type ="submit" value="     Submit     " >
</form>
</center>
</body>
</html>
```

The page has one form. It identifies the servlet program that will process the data, and it stipulates that data will be sent using the HTTP POST method. Password data would normally be handled by some more secure process than POST but we won't go into that here.

We included some rudimentary JavaScript with the page. If you open the page in your browser the JavaScript should run.

© OUDCE, January, 2013

Servlet code corresponding to this form is shown below.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Porterhousepwservlet  extends javax.servlett.http.HttpServlet {
//note that the name of the class matches the program name given in the form.

static Admin admin;  //This is supposed to be a reference to the system admin : Admin object
//we further suppose that initialisation routines within the servlet have set up this key system
//reference  when the servlet starts, but we don't show that here. Static objects exist at one
//unique point in the system, so we don't need to worry about synchronisation issues.

//You can think of servlets working in a callback mode. The form method was POST so
//data is sent to the doPost method. If the method was GET there would have to have been a
//doGet method. Both doGet and doPost override methods defined in the superclass.

public void doPost( HttpServletRequest rq , HttpServletResponse rsp)
                throws ServletException, IOException{

                rsp.setContentType("text/html");  //what sort of data we shall be sending back.
                PrintWriter ourOutput = rsp.getWriter(); //(standard Java used
                                                         //to write characters to streams).
                //now obtain the data supplied, note how we identify fields.
                String nominatedUser = rq.getParameter("username");
                String nominatedPassword = rq.getParameter("password");
                //We are storing data in variables associated with the method. These
                //are not instance variables of the object and are Java thread-safe.
                //We now check what has been sent with admin : Admin.
                //This is a variant on figure 6.2 for web interfaces. You are to
                //imagine that admin has a method called checkThisUser and that adapts
                //the situation in figure 6.2 and organises a call on checkLogin()
                If (admin.checkThisUser(nominatedUser, nominatedPassword)
                      setUpFirstPage(ourOutput);
                else setUpErrorPage(ourOutput);
                //if the user name and password are valid, setUpFirstPage generates an options
                // frame for the user. Otherwise, setUpErrorPage generates an error page.
                //The coding involves simple print statements like ourOutput.println("<html>");
                }

}
```

You should be able to see the annotation at the end of this little program explaining how the servlet generates the web page that is sent back as a response. This involves actually printing the page mark-up and content.

So, the actual page information is held in the programming part of this servlet – which is not a very robust way to do things!

We may have raised more questions than we have answered in this example but we hope that you get the general idea about how form data is obtained and how replies are organised. This particular servlet *could* be multi-threaded. For that reason, it cannot have instance variables. Had we stored the user name field data in an instance variable, there is a chance that we could process a user name relating to one user with the password relating to another one.

© OUDCE, January, 2013

The logic of this servlet is an expansion of what we previously showed you in Figure 6.2. It is to get the proffered user name and password details to admin : Admin by means of a form and a servlet means rather than by using Java event handling as described in Figure 6.2.

The process in Figure 6.2 involved admin : Admin organising the display of a Loan Frame. In the system we designed, transaction state information was implicitly encoded in the frames the librarian could see at any time, with privileged users being offered frames with fields relating to their special capabilities. Something like this would work as part of a web interface; but it is unlikely that we would permit loan processing over the web, and it is not really appropriate for admin : Admin to present *any* web page to an Internet user. This is why we have organised the page display from within the servlet. That servlet would send out frames that encapsulated what each particular user was able to do, and forms within these pages would target other Java servlets, and so on.

The analysis and design techniques we discussed in earlier chapters, which focused on use cases and developing objects from them, often seem to result in systems controlled by a single instance of an organising class and, as you see above, this is usually not appropriate in distributed situations. This particular problem arises because we have bolted an existing system onto a web interface.

There are other analysis and design methodologies that one might use if one were designing a web-based system from scratch; if one used them, the admin : Admin problem might not arise.

The simple servlet that we have just considered had no need to expend energy maintaining state; but that would be useful in a servlet controlling the manipulation of, for instance, a web shopping cart.

Java servlets offer several techniques for session tracking. One of the simplest to use involves **HttpSession** objects, where information is stored in the **key – value** pairing of an **associative store** (a simple indexing system widely used in computing). Data may be maintained over several HTTP transactions between the user and the server, although sessions usually expire automatically after some set period of inactivity.

Further issues in Enterprise Computing

We shall now outline *some* of the other technologies that are used for more sophisticated web applications. We might note that *each* of these is the subject of massive tomes of technical information. We are content with an extremely lofty overview!

The servlet we have just described made a decision based on simple database access, and presented a HTML page to the user on the basis of that decision. Servlets are normally employed to do more processing than that; but when a lot of database access and processing is needed, one would probably do it in a distributed location well away from the server and not use servlets at all.

The problem with this servlet is rather the reverse - it does almost no processing. The servlet generates the HTML of the page by using **println** statements to generate page tags - we did not really show that in the code but it is mentioned in the annotation. This is clearly a poor way to generate a static page (the error page would probably have completely fixed content and it would be foolish to embed that content within the code of a servlet).

© OUDCE, January, 2013

Java server pages (JSP)

Enterprise programmers would probably use JSP rather than servlets for the task we have just considered. JSP gets used when content is mainly static text and mark-up and only a little processing is needed.

JSP involves **dynamic web applications**. Microsoft has a similar dynamic page technology called **active server pages (ASP)**. The idea is the same in both of these – pages have programming in them and that programming is run at the server end and before they are served to the user.

This would be more appropriate in the situation just visited because one would produce the page first using a standard editor, then embed JSP processing within it.

Java server pages have the suffix jsp rather than html - you have probably seen them in your forays into the web.

When a user requests one, the web server recognises that processing is needed and strips the embedded code from the document. It then hands that code over to a servlet that it has created for the purpose[199]. The servlet runs the code as normal.

You can see that JSP is built on top of servlet technology. Web page counters often use JSP.

Here are some of the processing options available to Enterprise programmers.

Remote method invocation (RMI)

Remote method invocation permits a Java object on one machine to send a message to an object residing on another machine and obtain a reply. The recipient object is sometimes referred to as a **remote object**.

Genuinely distributed business applications might have a whole series of machines in the middle tier of figure 10.3 and those machines may well be considerable distances apart – certainly not all on the same LAN. Applications may well communicate using RMI in order to obtain services from each other.

RMI technology uses Java interfaces and Java serialization. A further important idea is that of a **directory service**. Each participating application knows how to talk to the directory service, which probably resides on a single machine. Objects wishing to send remote objects messages first locate those objects, using the directory service. If the object can be found the object wishing to send receives a local **stub** representing the distant object. The sending object transmits messages to the stub. The stub receives the message, packages it for transmission (a process known as **marshalling**) and sends the message, in serial form, to a counterpart stub running on the remote system (the counterpart is sometimes called a **skeleton**) this counterpart unpacks the data (**unmarshalling**) and sends it to the destination object. Replies are dealt with in the same way.

One disadvantage of RMI is that it is a **synchronous** communication technology. So, just as is the case with messages on a local system, messages block until they obtain their replies. This can be embarrassing in some situations but there are ways round it.

Directory and naming services are very important in distributed services. Naming services associated names with specific things, directory services associated a number of things with particular characteristics (find me a printer that does colour and handles Postscript, that sort of thing).

[199] The servlet may well be already running as a result of previous page accesses.

There is a Java technology called **JNDI (Java Naming and Directory Interface)** which facilitates the use of these services from within Java programs, and there are a number of international standards for directory services, of which the most important is probably **LDAP (Lightweight Directory Access Protocol)**. We have to content ourselves with simply mentioning these ideas and noting their importance.

Enterprise Java Beans

Enterprise Java Beans are small Java programs which provide particular services associated with the "business logic" (that is operational needs) of an Enterprise system. EJBs often model particular business constructs. For example, in an Enterprise solution to our Porterhouse system they might model Customer objects and in a typical web application they might model shopping carts.

EJBs are the ultimate way to separate processing from presentational or sessional concerns. EJBs "live" in a server program. One communicates with them by sending rmi messages. JNDI is an important tool for EJBs and they use it to locate such things as database services.

The EJB server provides the transaction management needed in distributed systems which pose problems of consistency, security and traceability.

Communication

Our final topic relates to information transfer. We might consider data flow from:

- humans to humans;
- humans to machines;
- machines to humans;
- machines to machines.

Communication was the impetus for the construction of both the web and the Internet. Reading data in one format and writing it in another is a recurrent problem in distributed computing.

Berners-Lee invented HTML as a language to hold and present data. At first, this involved scientific data, but you know how the language proved to be useful for very much more.

You will also know that HTML presents difficulties to programmers. HTML interpreters (browsers) forgive slack syntax and sloppy presentation but they do it at the cost of consistency; and slack syntax is one thing that machine processors are extremely poor at handling. On top of that, HTML has a fixed and pre-defined set of "tags". Particular applications find the HTML tag set limiting and inappropriate.

For all these reasons, another markup language called XML (Extensible Markup Language) was invented. This language had built-in extensibility (so you can define your own tags) and it was pre-set to be syntactically impeccable (so machines can process it easily).

XML has become the key way to structure data in Enterprise transactions. Enterprise Java provides built-in support for handling XML documents - programs which can extract the data from XML sources or generate them.

XML tags are based on plain text - so humans can read and make sense of an XML document[200]. XML documents are syntactically tight - so machines can parse them and extract data very easily.

[200] This takes a bit of practice!

XML has also become important in many non-distributed areas of computing too. At the time of writing, it is estimated that spreadsheets now hold more data than databases do. Even within a single PC data flows require a common approach to data formatting and presentation[201]. However, XML is not an efficient technology and its "flat" approach to data storage means that it will probably never replace more conventional relational databases.

XML is a language for defining mark-up languages. Each language defined from XML is an **XML application**. There are many such applications and some of them need associated processors to interpret the data.

XSL (the **Extensible Style sheet Language**) is one XML application widely used in distributed computing. XSL can transform one XML source into another. It is frequently used to process an XML document into a form that is suitable for direct display - such as XHTML[202].

SOAP (simple object access protocol) is another XML application that is much used in Enterprise Computing. It is a XML-based messaging service called that allows you to send data across the Internet using the HTTP protocol.

We cover XML in more detail in another document.

[201] The Project files controlling Java projects in Borland's JBuilder product are written in XML, for example and we believe that each new version of the Windows operating system makes more use of XML.
[202] XHTML is the XML version of HTML.

© OUDCE, January, 2013

References

These texts are either ones which the principal author used in writing this unit, or ones which, in our view, give cheap accessible accounts of the subject matter. You do not need to buy any of these. References described as "cheap" are available for under £15. Only one of these references is referred to directly (because we borrowed a table from it).

1. Bennet.S. Skelton. J. *Schaum's outline of UML*. McGraw-Hill. (ISBN 0-07-709673-8). This book is comprehensive, informative, and cheap.
2. Lunn K. *Sofware Development with UML*. Palgrave Macmillan. (ISBN 0-333-98595-8). Well-written and accessible, and not too "UML crazed!".
3. Tanenbaum. A. *Computer Networks*, Prentice Hall, (ISBN: 0-13-349945-6). One of the very few really great books in technical computing.
4. Tanenbaum, A. van Steen. M. *Distributed Systems – Principles and Paradigms*, Pearson Educational (ISBN: 0-13-088893-1). Authoritative and elegant, like everything Tanenbaum writes.
5. Various authors. *Enterprise Java 2 J2EE 1.3 COMPLETE*, Sybex (ISBN: 0-7821-4145-5). This is a clear and cheap introduction to Java Enterprise Computing.
6. Pooley, R. Stevens P. Using *UML: Software Engineering with Objects and Components.* Longman. (ISBN: 0-2016-860-1).

This is version five of a document that was first written in 2005. The picture on the first page is a fractal image associated with a mathematical recurrence. The recurrence exhibits extreme sensitivity to initial values, widely diverging patterns of behaviour, and local regularities – all reminiscent of the software development processes we have been discussing.

© OUDCE, January, 2013