Software Engineering

The Department of Computer Science, University of Cape Town

Software Engineering

by The Department of Computer Science, University of Cape Town

Publication date 2010

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Objectives

At the end of this chapter you will have acquired an introductory understanding of what software and software engineering are, as well as an understanding of some of the common myths surrounding the practice of software engineering.

Introduction

It could be argued that information systems are vital components of any civilisation. The human desire to record information goes back thousands of years to when humans first started painting on stones. However, it is the practice of recording and displaying information in a systematic manner that warrants the use of the term "information system". Such practices can be easily found in great civilisations, such as those of ancient Egypt, Greece and Rome.

The development and use of information technology (IT) has led to the birth of new generations of information systems. Computer-based information systems (software systems) have dramatically influenced our behaviour and the way in which we conduct every day activity. It is not surprising that the standing of any society in the world is now strongly linked to the level of penetration that software systems have in that society.

In this module we shall use term *software systems* to refer to information systems that contain or might contain software components. Although there are many information systems that do not involve computers, such as a card filing system of a small library or a manager's list of contacts, almost all modern information systems either use computers, or could use computers, to perform some of their functions (such as cash registers for point-of-sales processing, and stock control systems for small businesses).

Further, while we will occasionally mention information systems, this module is ultimately interested in the *software* components of an information system, and how to *engineer* software that can be reliably used by other people. Software is integral to computerised information systems. Without the underlying software, the system will not be able to do what its users intend, and if the software functions incorrectly, so will the information system.

Information systems and software

Software systems are made up of the following components:

- **Users** — the people who add information to the system, request information from the system, and perform some of the information processing functions.

- **Procedures** – the tasks performed by the human components of the information system.

- **Information** – meaningful data that the system stores and processes.

- **Documents** – manuals on how to use the system, sometimes even files of data which should not or could not be stored electronically.

- **Hardware** – not only the computers in the system but also any networks linking the computers, the input devices and output devices.

- **Software** – computer applications performing some of the system functions to record, process, and regulate access to some of the information worked with by the information system

Importantly, we need to consider what software is. Software is typically defined to be *instructions* that provide desired features, functions, and performance. They contain *data structures* which allow

the software program to manipulate the information contained in an information system. Importantly, software also includes documentation describing how the software performs the actions that it does, and how the software may be used. Notice that some of the documentation is for the software's users, while other portions of the documentation are for its developers and maintainers.

There are some important properties of software that you should consider when thinking of the discipline of software engineering.

- First, software is *engineered* rather than manufactured. Once the software has been developed, there remains no significant "manufacturing" process that could possibly lower the software's quality (i.e., introduce software errors, cause the software to deviate from what the customer requested, and so on). The cost of developing software lies almost completely in the engineering of the software, and not in the "manufacturing" of a "product" that customers can be hold in their hands.

- Software does not wear out with use, as hardware might. However, this does not mean that software does not *degrade* over time. A software program is continuously changed over its lifetime. If these changes occur too frequently, the bugs introduced by each change may slowly degrade the performance and utility of the software as a whole. Also, when software degrades in quality, there are no "spare parts" which can be used as replacements.

- Unlike hardware, most software remains custom built, rather than built using "off the shelf" components.

The importance of software engineering

Over the last few decades, software systems and the software that run them have become an important component to many aspects of our society, from commerce to medicine, engineering, the sciences and entertainment.

Importantly, the infrastructure of all developed countries rely heavily on software systems. Because of this, it is important that the software we use and rely on are of a high quality and fulfil our requirements of them. Gaining this high quality does not happen randomly or by accident — rather we need to engineer that quality into the software that we use.

When software fails, people may be bankrupted and even killed (consider safety critical systems which run planes, medical equipment, and so on). Because so much depends on software, software has become important to business and the economy. This means that the software engineer is always part of a larger environment, consisting of customers, other software engineers, managers, and so on. It is important that the engineer be able to interact appropriately with all of these individuals, and this, too, is part of software engineering.

Systems

Because information systems are what we build using software, it is important to consider exactly what it is that we are building: what is a system?

The word system is used regularly to refer to a coherent group of elements (or components) that together aim to achieve a certain objective, or to have a specific purpose. A more rigorous definition would describe a system as a group or combination of interrelated, interdependent or interacting elements forming a collective entity (a whole) with a control mechanism that helps the system achieve its goal.

Systems live in environments that are relevant to their existence. They receive inputs from the environment, and produce outputs for the environment. Systems tend to have a boundary that is defined in relation to the external environments in which they reside. The survival of a system relies on a control mechanism that regulates the processes of receiving inputs and producing outputs using feedback.

Figure 1.1, "A diagrammatic representation of a system" illustrates many of the important features of a system:

- There may be many different inputs

- There may be many different outputs

- The system is composed of interconnected components — the components may be of different kinds.

- Some of the outputs of the system are fed back into the system, so that the system can control itself and make corrective changes when unexpected or undesirable outputs occur

Figure 1.1. A diagrammatic representation of a system
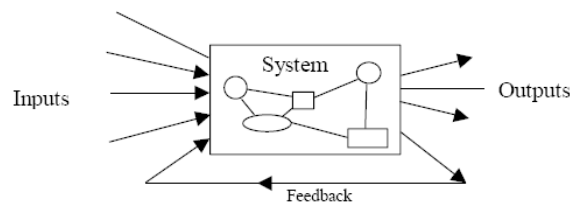
These components play a role in the software that we create: the inputs to the system will become inputs to the software. The outputs of the software will eventually be the system's outputs.

System boundaries

The smaller the system, the sharper its boundary. Large systems may have multiple boundaries as they interface with multiple systems. The boundary also depends on the point from which it is viewed in relation to other systems with which it interfaces.

An example of the complexity and difficulties of defining system boundaries is the ATM (automated teller machines). Consider the following candidates for system boundary:

- The physical machine itself

- The customer and the machine

- The machine and the bank's local database of customer accounts

- The machine and the bank's national network and central database of transactions and account balances

- The machine and the staff that loan and run regular software checks of the machine

Which of the above might be the "right" boundary for the analysis of the system in which the ATM is central? Different analysts may choose different boundaries, and these boundaries may change during talks with the customers wanting the system built, and with the ultimate end-users of the system. An important skill which systems analysts and software engineers must develop is to determine which boundaries are important to consider.

The software which we as software engineers develop are only a component of larger information systems in an organisation or group. This module concerns itself with the development and engineering of software, but to effectively engineer such software we must be able to understand the information system as a whole, and relate the software to it.

For effective and successful software engineering, software engineers have the difficult job of learning how the information system users perform their daily jobs, as well the tasks they attempt to achieve.

If the software does not support the users in their tasks, then the software will not be used, and the software development has failed.

American Federal Aviation Authority example

During the development of the American Federal Aviation Authority (FAA) automated flight control system software engineers tried to produce electronic counterparts for the flight strips — little bits of paper giving details of a particular flight which would be moved between operators and used for quick reference. This task proved to be impossible and the final system not only still has these bits of paper but the consoles have special slots to put them in.

This example illustrates an important point; that is any artificial system has a boundary which defines what is part of it and what is not. Sometimes this boundary is clear cut, say when the computer screen and keyboard are the boundary for a system, other times it is not, say when people and bits of paper form an integral part of the system. In the case of the FAA system some of the functions that were considered part of the system are solely performed by humans, but still within the boundary of the information system and the software being developed.

Categories of information system and software programs

Information systems

There are many kinds of software systems. Clearly, software used by each kind of system will differ. It is useful (and common practice) to break them down into categories such as the following:

• Data-processing systems

• Real-time systems

• Decision support systems

• Knowledge-based systems

Different people and authors may break down the categories further, or provide other categories. The list above is informal but useful, since (as we shall describe below) each category of system can be distinguished from other kinds of information system.

Data-processing systems

Such a system generally has some large database of information and the purpose of the system is to provide quick, easy access and processing of data.

Depending on the degree to which data is processed and analysed, systems may be classified as either transaction processing systems or management information systems. A system which basically manages the data necessary to perform the daily business is a transaction support system. A system which summarises data in a form useful for the management of a business is a management information system.

Real-time systems

The environment outside the boundaries of the system is not under the system's control. Therefore a system will need to be able to respond to data whenever it arrives — real time systems must respond quickly to changes in the inputs from the environment. Typical response times would be of the order of a few milliseconds or even microseconds. To achieve such a fast response the system needs to

prioritise its tasks, often dividing them into several processes that may interrupt each other. However, as soon as there are several tasks, they must be able to communicate properly with one another and not interfere with each other. Because of environmental interactions, real-time systems have to be robust to accidents, errors and failures in the external parts of the system. That is, they must respond in a safe and controlled manner in (almost) all conceivable circumstances.

A typical example is an auto-pilot that must adjust the engines and ailerons of an aeroplane to keep it on course. An automated manufacturing system — such as is used in car factories — has to detect when specific parts have arrived in designated zones of the factory, and then make sure that they are correctly assembled.

Decision support systems

Although a data-processing system may help to identify a problem in the business, it does not suggest any solution to the problem. In a decision support system, given a problem, the system attempts to fit the data on the problem into some model and thereby suggest a solution to the problem. A decision support system may have different models, say operational research or statistical models. The manager chooses the model appropriate to the problem.

Ultimately, of course, any decision is made by the manager and not the system. The manager may know or guess something which cannot be represented in the system's models, and this may affect their decision. The system is simply there to clarify the problem and suggest solutions as far as it is able.

Knowledge-based systems

In some situations it is not a large amount of data that needs to be handled, but a large amount of *knowledge*. *Knowledge* is a combination of rules, laws, constraints and previous experience. A knowledge-based system encapsulates a knowledge-base, like a database but filled with knowledge, and enables the user, possibly unskilled in the problem area, to use the knowledge-base to solve problems.

In business systems, the knowledge-based system often takes the form of what is called an "expert system". Expert systems embody the knowledge of a particular class of experts, such as medical doctors, and the system (ideally) provides the same answers as an expert of that class would. In the case of a medical expert system, this could be a diagnosis of an illness, and perhaps a recommendation for a specific choice of treatment or for further tests.

Software

Software programs can be categorised in the following manner:

- System software

- Application software

- Engineering / scientific software

- Embedded software

- Product-line software

- Web-applications

- Artificial intelligence software

System software

System software is software written to service and support other programs. Compilers, editors, debuggers, operating systems, hardware drivers, are examples of such software. Most system software

deals heavily with computer hardware, multiple users, concurrent operations and process scheduling, resource sharing and virtualisation, complex data structures and multiple external interfaces.

Application software

Application software are programs designed to solve a specific business need. Most software operating with business and technical data are application software, as are most real-time systems.

Engineering / scientific software

This software supports the use and production of scientific and engineering data sets. They are used in almost all engineering and scientific disciplines, from simulating water flow, lighting conditions, and aerodynamics, to examining the large scale structure of the universe. Engineering software is also used for design purposes (such software is called CAD software, for *Computer Aided Design*) and for automating the manufacturing of goods (CAM: *Computer Aided Manufacturing*).

Embedded software

This is software that resides directly within some hardware component in order to provide control over the component or other vital features required for it to function. Embedded software is widespread, and can be found in everything from phones and microwave ovens to cars, aeroplanes and medical equipment.

Web applications

A *web application* is an application accessed via a web browser over a network. Web applications offer a variety of functions, and some application software are now implemented as web applications, such as Google Docs.

Artificial intelligence software

Artificial intelligence software (AI) has been defined as "the science and engineering of making intelligent machines" (John McCarthy). Application domains that make use of AI software include robotics, expert systems, pattern recognition, theorem proving and game playing.

Legacy systems

While a lot of software that is in use is fairly current, state of the art software, companies often rely on software that has been in use for years, or even decades. This older software is called *legacy software*.

Legacy software is often, by today's standards, of poor quality: inextensible, convoluted, badly documented, and written in languages which are generally no longer used.

Worse, legacy software have often been continuously changed over decades to meet ever changing business and system requirements, contributing to the software's degradation.

Unfortunately, such legacy software is long lived precisely because it supports critical business systems. This makes it important that legacy systems be reengineered to remain usable in the future. One can think of this as a slow evolution of the legacy software. This evolution, these *changes*, are often done for one of the following reasons:

- The business is making use of a new computing environment or technology, and the software must be updated to support this.

- The software must meet new business requirements.

- As newer information systems, databases, and so on, become available, the software must be updated to be interoperable with them.

- Software must be modified to operate in a networked environment.

Changes to software are inevitable and are not unique to legacy systems. Software engineering must take this into account. This process of change is often referred to as *software maintenance*.

The benefits of software systems

The benefits of introducing new software are not always easy to identify. The person (or people) who are considering the introduction or extension of a software system (we shall refer to such a person as the "customer") may be very enthusiastic about the possible benefits of the new software, such as providing "better service" and "more control". However, given it is unlikely that the existing (manual or partly software-based) system is ruining a business, the advantages of the new software may be very difficult to quantify. In fact, unless the software is well-designed and properly thought out, it may bring no overall benefits at all.

The potential for significant benefits and the differences information systems have made in other organisations are an important motivation for many organisations to investigate the development of new or extended software systems in their businesses.

Organisations, their information systems, and the software they employ, are complex, and therefore the costs and benefits of information system and software development are hard to estimate and measure. Such systems are hard to analyse, design and implement. For this reason there has been much study into the relationship of information systems and software within organisations, as well as into information system and software development.

Broadly speaking, possible benefits can be divided into two categories:

• tactical benefits

• strategic benefits

Tactical benefits

Tactical benefits are ones which improve the day-to-day running of an organisation or group in measurable terms.

Customers almost always anticipate or require that new software and systems deliver cost benefits. In some cases software replaces an existing automated system, but the new software has cheaper hardware and lower maintenance overheads. For instance, where a single, centralised computer system is replaced by a network of smaller PCs. Or the software system replaces a manual system and so savings may be made on paper, paper storage space like filing cabinets, and office space. There may be an improvement in communications resulting in fewer telephone calls and faxes. In both cases there may well be a saving on the number of staff required to support the old system.

Another frequently cited benefits of software systems are their speed and accuracy. Information can be retrieved more quickly and with greater confidence in its accuracy. This can improve the productivity of employees. It may also improve the movement of goods and the supply of goods to customers. This could result in an organisation being able to handle more transactions and expand its business.

Strategic benefits

Strategic benefits are about improving the nature or abilities of a group or organisation. With a new software system, a business may be able to offer their customers new services. Or by examining the information held, could new customers or products could be identified. More speculatively, enhanced functionality might allow managers to quickly identify trends in sales or spending. This could lead to a competitive advantage in the market place.

With the introduction of expert systems, knowledge previously confined to a handful of individuals can be distributed and made available throughout a company. This could improve the functioning and performance of a company in many ways.

However, in many ways strategic benefits are even more difficult to quantify than tactical benefits, and without careful planning and design they will simply not appear.

The reasons for change

People do not usually embrace change merely for the sake of it. There is always risk involved in changing existing practices — to paraphrase an English saying "If it is not broken, do not fix it!".

The simplest reason might be that people like the look or sound of new technology and a company wants to be seen at the "cutting edge" of technology. Installing a new hi-tech software system shows the company off to good advantage. Companies can use this opportunity to reap some of the other benefits that a new system might bring.

Or a company may realise that new technology brings new processing facilities and methods within their reach.

More normally, companies develop new software systems as a result of external forces. Costs may be rising and new software could reduce overheads. There may be competitive pressures and unless changes are made to the business processes, a company will lose its competitive edge.

More mundane reasons for change includes new legislation requiring an update to business processes. A shortage of appropriate staff can also drive the need for new software.

Whatever the reason, once a company decides to install new software, it makes sense to maximise the return on the investment. Not only should the system address the problem at hand, but it should also bring as many other benefits as possible. It is this escalation of requirements which can transform a hi-tech business solution into a software catastrophe. We discuss this next.

Software myths

All people who come into contact with software may suffer from various myths associated with developing and using software. Here are a few common ones.

Management myths

Our company has books full of standards, procedures, protocol, and so on, related to programming software. This provides everything that our programmers and managers need to know. While company standards may exist, one must ask if the standards are complete, reflect modern software practice, and are — importantly — actually used.

If we fall behind schedule in developing software, we can just put more people on it. If software is late, adding more people will merely make the problem worse. This is because the people already working on the project now need to spend time educating the newcomers, and are thus taken away from their work. The newcomers are also far less productive than the existing software engineers, and so the work put into training them to work on the software does not immediately meet with an appropriate reduction in work.

Customer / end-user myths

A vague collection of software objectives is all that is required to begin programming. Further details can be added later. If the goals / objectives for a piece of software are vague enough to become ambiguous, then the software will almost certainly not do what the customer requires.

Changing requirements can be easily taken care of because software is so flexible. This is not true: the longer that development on the software has proceeded for, the more work is required to incorporate any changes to the software requirements.

Programmer myths

Once the software is written, and works, our job is done. A large portion of software engineering occurs after the customer has the software, since bugs will be discovered, missing requirements uncovered, and so on.

The only deliverable for a project is the working program. At the very least there should also be documentation, which provides support to both the software maintainers, and to the end-users.

Software engineering will make us create a lot of unnecessary documentation, and will slow us down. Software engineering is not about producing documents. Software engineering increases the quality of the software. Better quality reduces work load and speeds up software delivery times.

Review

Questions

Review Question 1

Read the following scenario about an online reservation system, then carry out the tasks below.

Global-Travel, an on-line reservation system

Global-Travel is an airline that sells all its tickets through an on-line system on the Internet. Prospective travellers register their details by filling in the form available on the company's web site. Once a registration has been confirmed as valid, the travellers can proceed by selecting the destination to which they wish to travel, the date of travel, and the type of ticket they wish to purchase. Only those of destinations to which Global-Travel flies appear on the list of destinations for customers to choose from. Once a destination has been selected, the date box displays only those days of the week on which Global-Travel flies to the selected destination.

Once all ticket details are entered, the customer submits the details to the systems by clicking on the appropriate button. The system responds by either accepting the reservation or rejecting it. If a reservation is rejected, the system displays the reason (e.g., no available places on the specified date or no places available on the specified date for the selected type of ticket). The customer may then choose to change the date of travel, upgrade his/her ticket or abandon the reservation.

If a reservation is accepted, the system prompts the traveller to proceed to the payment section to purchase the ticket or reserve it for 24 hours. Global-Travel web site has a secure server and accepts payments by most credit and debit cards. Once the traveller completes the payment section, the system displays the details of the purchased ticket and requests the traveller to check the details and confirm or abandon purchase by clicking on the appropriate button.

The marketing department at Global-Travel has another information system that is linked to the online site and monitors sales of tickets on all flights. It uses this information to frequently display various promotions and special offers on the site. The accounting department system is also linked to the online site and receives all payment information.

Review question tasks - Answer the following questions based on this scenario:

• Describe the context of Global-Travel online sales system.

• Describe the collection of information that is relevant to that context.

• Specify who has access to this information.

A discussion of this question can be found at the end of the chapter.

Review Question 2

Do the following, based on the Global-Travel scenario:

- Describe the inputs and outputs of the system.

- From the point of view of a prospective traveller, describe the feedback loops in the system and how this affects the input and the output of the system.

A discussion of this question can be found at the end of the chapter.

Review Question 3

Read the following scenario about an automatic train system, then carry out the tasks below.

ARTC, an on-line reservation system

A railway authority have been asked to fit all its trains with an automatic signalling control system within five years. The main objectives of the project was to increase the safety of rail travel by:

- reducing the number of trains that passes through red signals or possibly preventing trains from passing any red signal.

- reducing the number of train accidents that result from head on collision or moving trains colliding with stationary ones.

The "Automatic Railway-Train Control System" (ARTC) system should alert the train driver when the train is approaching a red signal. The alert of a red signal must take the form of audible sound and visible red light in the driver's cabin. The alert must start at a specified distance from the red signal (called the *red signal alert point*), and at the very least the light must continue to be on until the signal switches back to green. The alert sound must also stay on until the driver switches it off. The switching off of the light is used as an acknowledgement that the driver has heard the alert. The level noise from the alert sound increases until it gets switched off.

The ARTC system checks and records the train speed each time it passes a red signal alert point (RSAP). If the train speed was higher than it should be at this point, the system should alert the driver to slow the train down.

The action taken by the driver after his/her train reaches a RSAP depends on the system that would be installed. Currently there are three different systems available.

The first, called *Fully Automatic Railway-Train Control System* (FARTC), takes over the breaking and the driving systems of the train after it passes a RSAP and gradually brings the train to a halt before the red signal.

The second, called *Semi-Automatic Railway-Train Control System* (SARTC), does the same as FARTC but has to be triggered by the driver and can be over-ridden also by the driver.

The third, called *Automatic Alert Railway-Train Control System* (AARTC), simply alerts the driver of the red signal and expects him/her to stop the train before the signal.

The choice of system will depend on the budget and the time-scale allowed.

Tasks

Answer the following questions based on this scenario:

- Describe the possible boundary of each of the proposed system.

- Discuss how they differ and why.

A discussion of this question can be found at the end of the chapter.

Review Question 4

Do the following, based on the ARTC railway scenario:

Describe some of the control (feedback) mechanisms that were mentioned in the case study and their function. State the inputs and output of these mechanisms and how the feedback process could affect them.

A discussion of this question can be found at the end of the chapter.

Review Question 5

Describe what is meant by an information system?

A discussion of this question can be found at the end of the chapter.

Review Question 6

Describe one example from your own experience of each of the following types of systems:

- Real-time systems

- Data-processing systems

- Decision-support systems

- Expert systems

A discussion of this question can be found at the end of the chapter.

Review Question 7

The United Kingdom has been debating whether to join the single European Currency for a while now. What might be some of the consequences to organisations whose business is the export of hand-made furniture to EU countries and the USA? How would these consequences influence decisions about software and information system development?

Legacy systems are old software which continues to be used today, even though continual, ad-hoc updating of the software has introduced many bugs and inconsistencies. What are the potential pitfalls to keep in mind when updating software in order to avoid producing (in the long-term) inefficient software that we would consider to be legacy software?

A discussion of this question can be found at the end of the chapter.

Review Question 8

A seminal work on software engineering is Fred P. Brooks's paper, "No Silver Bullet — Essence and Accident in Software Engineering".

An important concept that he tells us is that:

> Fashioning complex conceptual constructs is the *essence*; *accidental* tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.
>
> —Fred Brooks

In the above quote, Brooks is arguing that our ability to manage the direct tasks of the programming and production of the software itself is no longer a major problem: tools and techniques exist to help

us handle this. *The* problem of software engineering is in managing the conceptual aspects of it: the software's specification and design, and the testing of this specification and design.

We cannot expect tools to automatically manage or drastically lesson the conceptual problems related to software engineering: there is no "silver bullet", no tool that will solve it all; there is only sound engineering practice.

This paper is available online in an abridged form. Find a copy and read it.

There is no discussion section for this review question.

Answers

Discussion of Review Question 1

Describe the context of Global-Travel online sales system.

The system is in the context of providing a service to potential passengers, providing a communications medium for messages from the marketing department and providing booking, reservation and payment data to the sales function of the company.

Describe the collection of information that is relevant to that context.

Information is collected from the customer, the marketing department, the seat availability database, and the credit reference system.

Specify who has access to this information.

The customer has access to information fitting their request. Sales has access to booking, reservation and payment information. Marketing has access to the sales information and the seat availability, on which to base its promotion decisions.

Discussion of Review Question 2

Describe the inputs and outputs of the system.

Inputs from the customer include:

• Flight requirements

• Reservation/booking decision

• Payment requirements

Inputs from the marketing department include:

• Offer availability messages to display

Inputs from the sales department include:

• Seat availability

• Payment acceptance decisions

Output to the customer include:

• Details of available flights meeting requirements

• Request for booking/reservation decision

• Request for payment

- Payment acceptance decision

- Display of offers (as suggested by marketing)

Outputs to the marketing department include:

- Details of sale information

Outputs to the sales department include:

- Customer's flight requirements

- Customers booing/reservation details

- Customers payment details

From the point of view of a prospective traveller, describe the feedback loops in the system and how this affects the input and the output of the system.

The system displays only information about flights meeting the traveller's requirements. If the requirements change, the system responds with new suggestions.

If there are seats meeting traveller's requirements the system requests a booking/reservation decision.

The system responds to it information on the validity of the payment offered by the user.

Discussion of Review Question 3

Describe the possible boundary of each of the proposed system.

The boundary of FARTC is simply the system's interaction with the RSAP, the braking system itself and some method for the system to know how hard the brakes need to be automatically applied.

The boundary of SARTC needs to include the driver, since the trigger by the driver will make the SARTC system start to break the train. The system must also include the braking system to be applied and the RSAP signal.

The boundary of AARTC should include the RSAP and the driver. Since this system does not make the brakes come on, there is no need to include the braking system as part of this information system.

Discuss how they differ and why

The first system does not need to include the human driver — all that is required is a signal requesting that the noise stop playing.

The second system must include both the driver and the braking system, since the driver needs to engage the SARTC system, and can also provide the input of overriding the system.

The final system does not include the braking system at all, since it is simply an alarm system to be switched off by the driver.

Discussion of Review Question 4

The FARTC system has three main feedback loops:

- The RSAP signal input triggers the brake

- When the train is stationary, we might assume the brake mechanism releases

- The driver switched off the light

The SARTC system has five main feedback loops:

- The RSAP signal input triggers the warning

- The driver triggers the system to start braking

- The driver can trigger the system to stop braking

- When the train is stationary, we might assume the brake mechanism releases

- The driver switched off the light

The AAARTC system has two main feedback loops:

- The RSAP signal input triggers warning

The driver switched off the light

Discussion of Review Question 5

We might define an information system as a system of interrelated elements working together to achieve some goal, composing of:

- A context,

- A collection of information that is relevant to that context, and

- System functions to record, process, and regulate access to the information.

Obviously in this module we are most interested in information systems that are comprised chiefly of software components.

Discussion of Review Question 6

Real-time systems: Examples should be systems that respond to changes in very short times. Examples might include:

- automated braking systems on cars (that detect jamming and unlock the breaks for a fraction of a second)

- automatic pilot systems (that detect undesirable plane behaviour and alert the human pilot)

- computer games (detecting joystick movements and changing the display in fractions of a second to give the effect of movement)

- An autonomous robot fish (that swims in a river filtering poisons, avoid objects and surfaces to recharge its solar cells

Data-processing systems: Examples should refer to systems that process large amounts of data, and possibly provide communication with some larger network of computer-based systems. Examples might include

- Point of sales terminals that process (and validate) credit card transactions

- Electronic mail systems to allow employees to communicate with each other in different rooms or buildings

- Booking systems, such as an air-plane reservation system whereby a travel agent places a reservation for a seat into the records for a particular air-plane company, to prevent other agents booking the same seat

Decision-support systems: Examples should be about systems that attempt to analyse data in terms of certain models, and perhaps predictions of future outputs based on extrapolations of the data. Examples might include:

- stock re-ordering systems, that monitor stock changes and attempt to predict optimal ordering to reduce stock keeping costs but ensure orders can be met without delay

- marketing systems that model likely demand and sales for new products based on timing and advertising budget decisions

Expert systems: Examples should be about systems that model human expertise and decision making. Examples might include:

- expert systems that process inputs against a knowledge-base of past situations and decision rules to make (and justify) suggestions for decisions (such as diagnosing illnesses or categorising applicants for life insurance or loan applications)

- a car-diagnosis system to guide a telephone centre operator to ask questions about a car to locate the problem so the right parts can be taken along by a rescue vehicle

Discussion of Review Question 7

There are a number of consequences, including:

- currency changes (EU countries would no longer need any change in currency, while the currency to be exchanged with the USA customers will be different)

- legal changes — there may be new or changed import/export laws between the UK and EU countries, possible less or different taxation

Implications for software system are that any existing systems will need to be changed. It might be that the company has a sister-company elsewhere in the EU, so perhaps a decision about which information system (or which bits of each) should be retained so that both companies will move towards using the same system. The jobs done by staff for processing EU sales will probably be simpler (and different) from the processing of USA sales, and the human and computer information system needs to be adapted to make such different order processing straightforward.

An important thing to consider when updating software is that the changes are properly engineered: while all changes to software can introduce inconsistent behaviour, inefficient behaviour, and bugs, when bad software engineering practices are followed this is much more likely to occur.

Chapter 2. Process and Model

Objectives

At the end of this chapter you should be able to:

- Define software engineering.

  Describe generic framework activities of the software engineering process.

- Describe various process models, such as the waterfall and prototyping models, in depth.

- Explain the difference between prescriptive and agile process models.

- Describe the main components of CASE tools, and how they can address system development problems.

The software crisis

There were many difficulties in the development of large software systems during the 1960s and 1970s. The term "software crisis" dates from that time. The problems stemmed from an inability to apply the techniques used to build small software systems to the development of larger and more complex systems. The typical way to develop small systems can be described as "code-and-fix".

The code-and-fix approach to software development

The "code-and-fix" approach to software development is not a proper life cycle (see later this unit). Code-and-fix development occurs when software engineers come together with a vague set of requirements and start producing software, fixing it, and changing it until the correct product appears.

Figure 2.1. The code-and-fix approach

```
┌──────────────────┐
│ Programmers think │
│ requirements are  │
│    understood     │
└──────────────────┘
         │
         ▼
┌──────────────────┐
│  Code part of the │
│      system       │
└──────────────────┘
         │
         ▼
┌──────────────────┐
│  Fix any errors and│
│ enhance if required│
└──────────────────┘
```
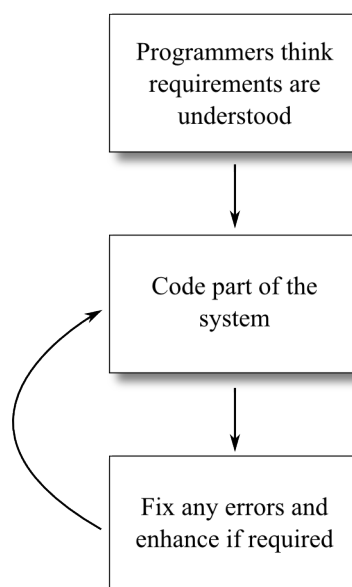
This is the simplest way to produce software and is invariably how every programmer learns to program. But for anything other than small software projects, code-and-fix is a disaster for a number of reasons:

- There is no way to estimate time-scales or budgets.

- There is no assessment of possible risks and design flaws: you may come close to a finished product only to find an insurmountable technical problem which sets the whole project back.

We only mention the code-and-fix approach in the context of life cycle models since it is a base-line model which we should avoid. From a software engineer's point of view, code-and-fix is a worst case scenario.

Many project failures resulted from the inability to scale the techniques employed when developing small software systems to handle larger, more complex systems. This failure leads to:

- never completed systems

- missed deadlines

- exceeded budgets

- a system that does not do all that is required of it

- a system that works but is difficult to use

- a system difficult to modify to meet changes in organisational needs and practices

- a loss of trust from users, who may experience many problems with using the software.

These problems were largely due to the lack of any framework for the planning and organisation of software development projects. Although some software projects were organised, and these were often the more successful ones, it was the luck of the draw whether a project manager had good intuitions for software development, and whether or not problems arose due to misunderstandings between the customers and the developers of the system. Likewise, there were no clear methods to monitor whether a system was soon to go over budget or miss deadlines.

From some of these problems we can see that at some stage the system developers attempted (not always successfully) to understand the requirements for the new system. We can now include in our diagram of the process these specified requirements for the new system:

Figure 2.2. The process with requirements
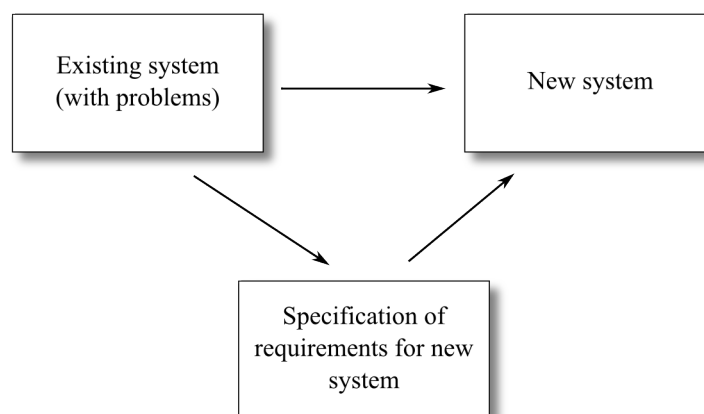
Software engineering and the software process

Recognising these problems, work was carried out to understand the process of software development and to transform it into a **reliable and rigorous discipline**, like architecture or engineering. An

improved **process** should produce software that is **correct**, **reliable**, **usable** and **maintainable**. By understanding the process, it should be possible to **plan projects** with more accurate predictions of **cost** and **time**, and provide ways of **monitoring** intermediate stages of project progress, to be able to react and re-plan if a project begins to go off budget or timescale.

Software engineering is exactly the discipline of producing such software. Fritz Bauer defined software engineering to be: *"the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."*

Much research has been put into the study of past systems that were both successful and unsuccessful. This can be summarised as:

- Some software development activities appear to be common for all successful projects.

- Some activities need to occur before others.

- There is a need to both understand that requirements change, and to manage this change.

- Any existing systems need to be understood *before* working on the design of a new one.

- It is wise to delay decisions that will constrain the final system — this can be achieved by initially designing an **implementation-independent logical design** (see the next chapter), before committing to a detailed design for a particular physical set of hardware and software.

Analysis of such findings led to a model of what is called the **software process**, or **system life cycle**. The *software process* is the process of engineering and developing software; a **process model**, or **life cycle model** is a *descriptive* model giving the best practices for carrying out software development (i.e., for carrying out the software process). However, a process model is often treated as a *prescriptive* model that needs to be followed precisely, without any deviation. This should not be the case. The specific model of the software process used should be tailored to meet the specific needs of the project and the developers working on the project.

Note

The phrases "Software process", "Software Life Cycle", "System Development Life Cycle", "System Life Cycle", "Development Life Cycle" are all used to describe the same concept.

The layers of software engineering

Software engineering is a discipline that can be pictured as being built up of layers (Figure 2.3, "The layers of software engineering").

Figure 2.3. The layers of software engineering
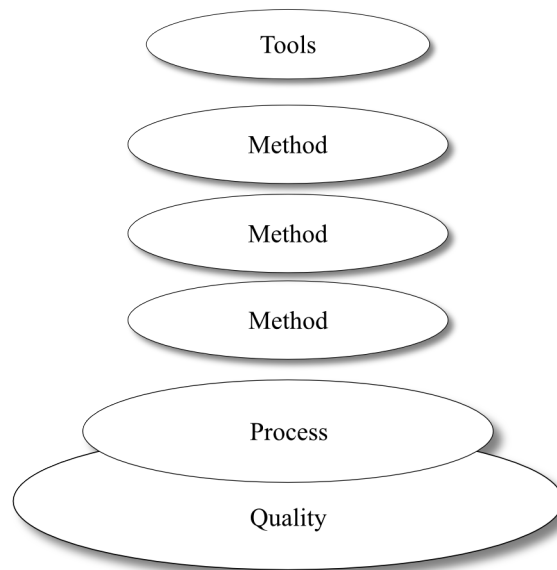
Software engineering demands *a focus on quality*. This should permeate throughout the rest of the engineering discipline.

On top of this comes the foundation of software engineering: *the software process*. The process is the framework on which the rest of software engineering is built. The process defines how management occurs, what the required input and output products are, what milestones should be reached, and so on. The process also describes how quality should be ensured.

On top of process, software engineering consists of *methods*. These describe how the various portions that make up the software process should be carried out. For instance, how to communicate with clients, how to test the software, to gather requirements, and so on. This makes up the process model.

And above all of this, and in support of the whole discipline, are the *tools*. The tools support the software process. Such tools are called *computer-aided software engineering* tools.

A generic framework of the software process

A software process consists of the activities that are carried out during the development of every software system. There are specific activities which are carried out at specific times, as well as activities carried out throughout the project's lifetime. Such life-long activities are called *umbrella* activities.

A generic framework defining these activities for the software process can be given. It identifies activities common to most of the models of the software process, although each model adapts the activities to its own ends.

The activities are as follows:

- **Communication** - This activity involves the gathering of software requirements from the customer, and related sub-activities.

- **Planning** - This is the activity of planning the work required to develop the software. This includes risk management, listing the associated outputs, and producing a schedule for the work.

- **Modeling** - This activity is involved with modelling both the requirements and the software design, so that both the developers and the customers can better understand the work being carried out.

- **Construction** - This is the development of the software. This activity also includes sub-activities for testing the software.

- **Deployment** - The software is delivered, and the customer provides feedback on the software.

Software models

The framework just presented provides a list of generic activities common to most models of the software process. However, each model treats the activities differently, and each model is suitable for different projects and for different teams.

It is important to realise that the activities outlined in the process models given below *should* be modified, based on:

- The *problem* having to be solved.

- The *characteristics* of the project.

- The nature of the *development team*.

- The *organisational culture*.

Prescriptive and agile models

Prescriptive software models are those which *prescribe* the components which make up a software model, including the activities, the inputs and outputs of the activities, how quality assurance is performed, how change is managed, and so on. A prescriptive model also describes how each of these elements are related to one another (note that in this sense, "prescriptive" is not meant to indicate that these methods admit no modification to them, as we previously used the word).

On the other hand, *agile software models* have a heavy focus on *change* in the software engineering process. Agile methods note that not only do the software requirements change, but so do team members, the technology being used, and so on. We will discuss agile methods later in this chapter.

Prescriptive software models

The waterfall life cycle model

The waterfall model was the first, and for a time, the only process model. This model is also known as the "traditional" or "typical" software life cycle.

> ## Note
>
> Some writers use the acronym TLC, standing for "Traditional Life Cycle". In this module we will refer to this software model as the "waterfall" model. This model is sometimes also called the "linear sequential" model.

The features of the waterfall model are as follows:

- The system development process is broken into distinct stages.

- Each stage involves a particular project activity — such as "communication" or "construction".

- Each stage, when completed, results in a deliverable (also called a product).

- The input to any particular stage are the deliverables from the previous stage.

- The model presents the stages in a strict, one-way sequence — a project cannot go back to repeat a stage once that stage has been completed. Any required re-work (as a result, for example, of changing software requirements) is very limited.

The name "waterfall" comes from likening this method to a river cascading over a series of waterfalls: the river is the output from each stage; the output from one stage is the input to another, in strict-sequence, and at no point do the stages reverse; a project cannot go back to a stage that has previously been completed.

Figure 2.4. The waterfall method
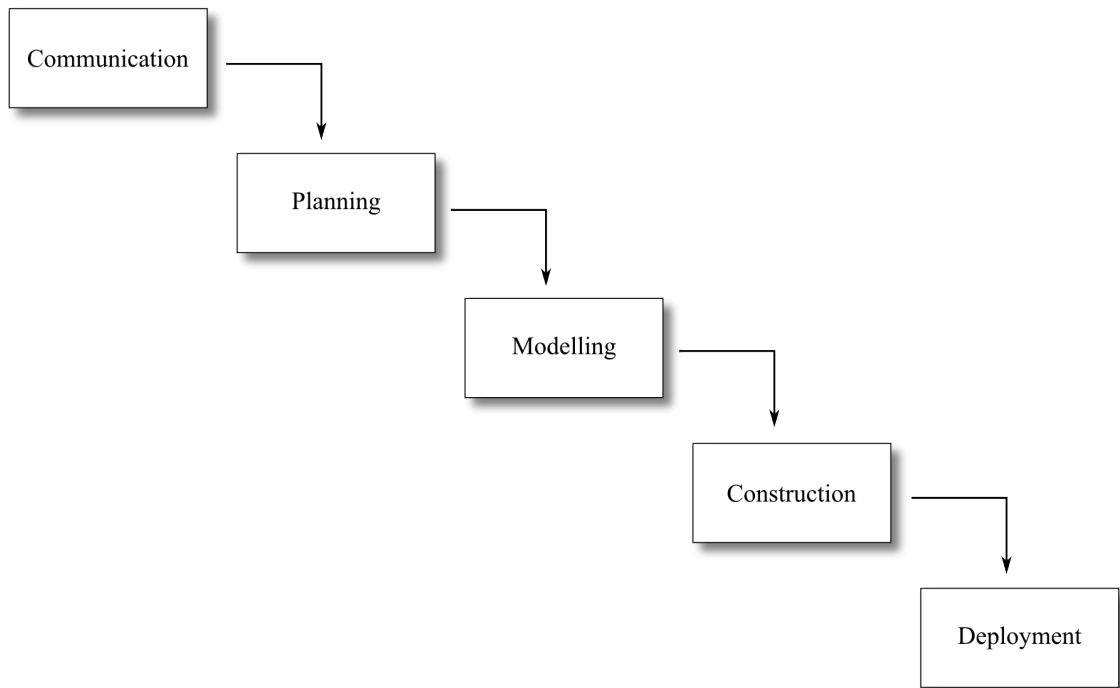
Each stage of the waterfall method flows into another.
Stages do not flow backwards through the model.

There are some slight differences in the way the waterfall model is presented between different books, however these differences are usually related to the number and names of the stages. Any presentation of the waterfall model will present a very similar sequence of stages to the following:

- Communication

- Planning

- Modelling

- Construction

- Deployment

There are a number of advantages to the waterfall model:

- The stages consist of well-defined tasks which promotes good scheduling and cost estimation (if all stages occur in the expected sequence once only).

- The deliverables provide targets or milestones to see how far a team has reached in the development process.

- The life cycle is broken into well defined stages — so staff expertise can be used efficiently (e.g., a data modeller only needs to work on certain stages, a programmer only on other stages, and so on).

- At any one time the project team knows what should be happening and the deliverable(s) they are to produce.

However, there are also a number of major limitations of the waterfall model, which occur frequently in software development:

- It is rare that a software development project will follow the sequential process that the waterfall model uses.

- Although the requirements are specified early on, user understanding and feedback of the software will not occur until *after* the system is implemented, which is possibly too late (or very costly) to change.

- The user may not be able to describe the requirements of the desired system in any detail early on.

- The model does not easily allow for the anticipation of change — some systems take years to develop, but once the early stages have been completed the model commits the project to a fixed specification of the system.

- Many projects based on the waterfall model stress the importance of certain products (documents) being delivered at certain times — it is possible for a project to become managed in a bureaucratic way, with documents being delivered on schedule, but the focus drifting away from developing a usable, effective system for the users.

- If a problem is identified at a later stage, the model does not make it easy (or cheap) to return to an earlier stage to rectify the mistake (since all intermediate steps will need to be repeated, resulting in significant, unplanned, time and resource costs).

For many development projects, the limitations of the waterfall model are usually considered to far outweigh its advantages.

Incremental process models

Incremental process models provide limited functionality early in the software's lifecycle. This functionality is then expanded on in later releases. We will examine two such processes.

Incremental development software model

Incremental approaches attempt to maintain some of the advantages of the pure waterfall model, but attempt to allow for greater change management and overall flexibility in the software process.

Figure 2.5. The incremental development software model
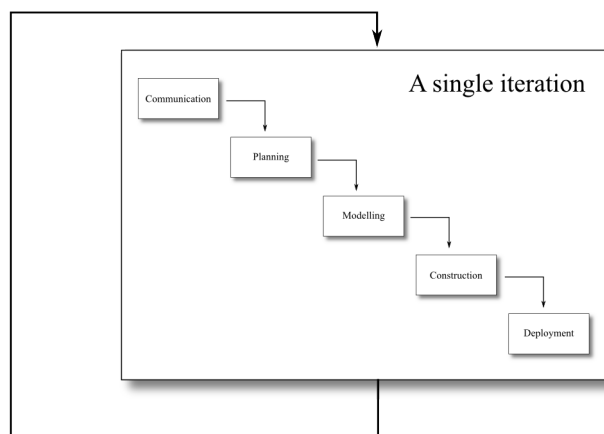
Software development occurs in small increments, allowing this
model to handle change far better than the waterfall method.

The incremental model allows the developers to quickly release a version of the software with limited functionality, and then at each development iteration to add additional, incremental functionality. The development in each iteration occurs in a linear method, as with the waterfall model. Ideally, the most important functions are implemented first and successive stages add new functionality in order of priority.

By doing this, the full development task is broken down into smaller, more manageable portions, allowing implementation problems to be highlighted before the full system is completed.

Incremental delivery is this process of releasing the product to the client at the end of each iteration. This allows the client to use regular, updated versions of the software, giving them the capability to judge the progress of the software development.

Although early, incremental delivery of the software is an option for project managers, it is not necessarily the case that each sub-system is delivered to the user as soon as it is completed. Reasons for delaying delivery may include the complexities associated with integrating the customer's existing software system with the limited functionality of the new system — it might make better sense to wait until a more functional implementation of the new system is completed.

An incremental development approach has the following advantages:

- The process is more responsive to changing user requirements than a waterfall approach — later sub-systems can be re-specified. Also a modular approach can mean maintenance changes are simpler and less expensive.

- There is an opportunity for incremental delivery to users, so the users can benefit from parts of the system development without having to wait for the entire life cycle to run its course.

- Incremental delivery means that users have a portion of the software to examine in order to see how well the software meets their needs, and whether the software requirements have to be modified.

- Complete project failure is less likely, since users will have some working sub-systems even if time and money run out before the complete system is delivered.

- The project can begin with fewer workers, as only a subset of the final product is being worked on.

- The risk associated with the development of the software can be better managed.

- The time taken to develop previous iterations can be used as an estimate for the time needed to develop the remaining iterations, and hence improve project planning.

There are some costs, and dangers associated with an incremental development approach though:

- This development model relies on close interaction with the users — if they are not easily available or slow in evaluating each iteration, the whole process can slow down.

- The reliance on user involvement can exacerbate the already difficult task of estimating the amount of time and budget required.

- High user involvement means that resources are drawn away from the customer's normal operation during system development.

Rapid Application Development (RAD) process model

Rapid Application Development is an incremental process model that has a focus on short development cycles (hence the term "rapid"). This speed is obtained by using off-the-shelf components, and a component-based design and implementation approach.

It has the following advantages:

- Development cycles are rapid, typically between 60 to 90 days.

It has the following disadvantages:

- For large projects, RAD may require a large number of people to split the project into a sufficient number of teams.

- The developers and the customers must be committed to the necessary activities in order for the process to succeed.

- The project must be suitably modularised in order for RAD to be successful.

- RAD my not be appropriate where high-performance is necessary.

- RAD may also not be appropriate when technical risks are high.

Evolutionary process models

Product requirements may change with time, even while the software is under development. Worse, the initial specifications may not be detailed, and tight deadlines may result in a need to have software quickly ready.

All of this points to a product that evolves over time, and evolutionary process models are designed to satisfy the engineering requirements of these products. Evolutionary process models are, as we shall see, iterative; they allow for the software engineer to deliver a product, and then iteratively move towards a final product as the understanding of the product improves.

We will discuss two such process models below. One disadvantage to keep in mind is that it can be difficult to plan the number of iterations, and hence the length of the project, in advance.

Prototyping life cycle model

A prototype system is a smaller version of part(s) of the final system that gives the user a sense of the finished system's functionality. It has some of the core features of the final system and, where features and functions are omitted, it pretends to behave like the final system. Prototypes are typically developed quickly, may lack unnecessary features, may be buggy, and have poor usability. However, prototypes can fill an important role in understanding software which does not have clear requirements.

Where the system to be developed is a truly new system, there may be no clear requirements defining the software's behaviour. By building a prototype, both the developers and users have some real, visible working system model on which to focus their ideas. An analysis of this prototype forms the basis for the requirements specification, and perhaps even some of the design. If there is still uncertainty of the new system and questions still remain, further prototypes can be developed (or an existing prototype extended). In this way, prototyping allows developers and customers to better understand incomplete and fuzzy software requirements.

Once the developers and users have a clear idea of the software's requirements, the project can move into a another development life-cycle, and the prototypes are *thrown away*. This is important, since as we previously mentioned, the prototypes are generated quickly and are not designed to be robust or complete.

To prototype quickly and effectively, fourth generation languages (4GLs), graphical user-interface (GUI) tools (like those that come with Visual Studio, QT and GTK), and off-the-shelf components are commonly used. The quality of the prototype is only of concern where it would hinder the prototype's use in understanding the final software being developed. If the prototype is usable enough to meet the objectives put forward for its development, the prototype has been successful.

A diagram of the disposable prototyping life cycle stages is presented as follows:

Figure 2.6. Disposable prototyping
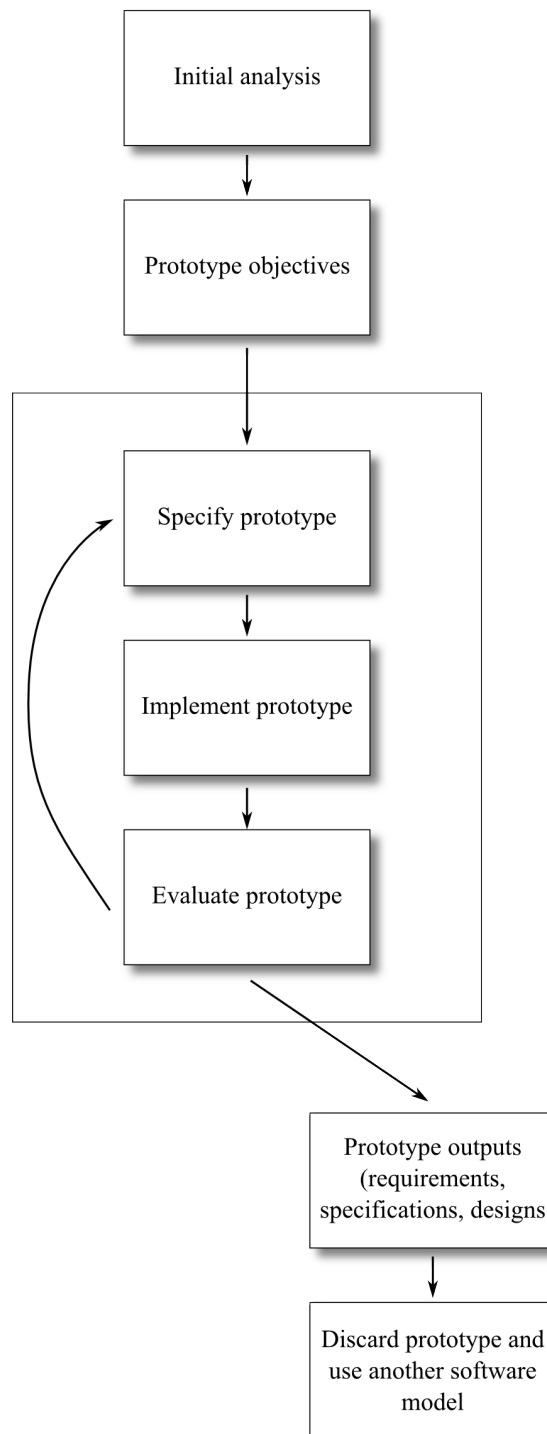
As can be seen, after some initial analysis a set of objectives is developed for the prototype. These objectives may differ between projects — perhaps detailed requirements need to be elicited, perhaps alternative user interactions are to be evaluated and so on. Each version of the prototype should be specified so that the software can be correctly designed and implemented — a prototype that does not fully test the objectives is a waste of resources, and may be misleading. Once a prototype has been completed it should be evaluated against its objectives. The evaluation decides whether the prototype should be extended, a new prototype developed, or — if the specified objectives are met — if the project can move on to develop the software using another process model.

Advantages of prototyping include:

- Users get an early idea of the final system features.

- The prototype provides an opportunity to identify problems early and to change the requirements appropriately.

- The prototype is a model that all users and customers should be able to understand and provide feedback on, thus the prototype can be an important tool to improve communication between users and developers.

- It may be possible to use aspects of the prototype specification and design in the final system specification and design, thus some of the prototype development resources can be recouped.

A major problem with developing "disposable" prototypes is that the customer may believe it to be the final product. Customers may not understand the need to re-engineer the software and restart development, and may ask that the prototype be "cleaned up" and released to them.

Boehm's spiral model

The *spiral model* was published by Barry Boehm in 1986. It provides an iterative, evolutionary approach to software development combined with the step-by-step aspects of the waterfall process model and the requirements analysis abilities of prototyping. It is intended for development of large, complicated software projects.

This process model provides for the rapid development of progressively more complete versions of the software. Each iteration of the evolutionary development will have a release, which may merely be a paper model of the software, a prototype, or an early iteration of the software.

Each iteration of the spiral model contains all of the activities from the generic process framework outlined above: communication, planning, modelling, construction and deployment. One can consider an iteration to be an arc in a spiral: each arc contains the same breakdown of how the development is approached, but each arc will focus on something new.

Each iteration also requires a certain amount of risk assessment, in order to lay out the plans and determine how the project should proceed. Risk assessment will adjust the expected number of iterations, and also affect what milestones are expected. The development of prototypes (as with the prototyping life cycle model) is an ideal way to mitigate the risks involved with poorly understood or vague software requirements.

The advantages of this model are:

- The spiral model considers the entire software life-cycle.

- Because of its iterative approach, it is adaptable, and appropriate for large-scale projects.

However, the model does have disadvantages:

- It requires expertise at assessing and managing risk.

- It may be difficult to convince customers that such an evolutionary approach is necessary.

Component-based development

In this process model, software is developed by integrating pre-developed software components and packages. This may be commercial, off-the-shelf components, or they may be components previously developed by the software engineers themselves.

Each component needs to present a well-defined interface to allow for easy integration.

The component-based model proceeds through the following steps:

- Determine what components are available and evaluate them for their suitability.

- Consider how the component will be integrated with the software.

- Design the software architecture so that the components may be easily employed.

- Integrate the components into the architecture.

- Test the software to ensure that all of the components are functioning appropriately together.

This approach may lead to a strong culture of component reuse. It has been shown that this model also leads to a 70% reduction in development time, an 84% reduction in project cost, and increased developer productivity.

This model is similar to RAD, which we discussed earlier. Note that RAD differs in that it is focused on *rapid* development, rather than specifically on component reuse.

The formal methods model

The *formal methods* model focuses producing formal, mathematical specifications of the software product. When the software is built to the given specification its behaviour will already have been verified to strictly meet the software's specific requirements.

The importance of this model comes from its ability to discover ambiguity, incompleteness, and inconsistency in the software requirements. This stems from the formal, rigorous, mathematical approach employed for software specification.

Formal methods are important to the development of safety-critical software, such as that used in aircraft avionics and medical devices. Formal methods have also been employed in business-critical software, where, for instance, severe economic problems may occur if the software contains errors.

Even though this model can produce extremely reliable software, it has many disadvantages:

- The development of the software's formal model is both time consuming and expensive.

- Very few developers have any training in formal methods, and so require extensive training.

- Formal methods cannot easily be used as a means of communicating with the customer and with non-technical team members.

Remember that expert training in formal methods is needed to employ this process model. Formal methods also do not replace traditional methods: they should be used in conjunction with each other. Importantly, employing formal methods does not mean that the software developer need not adequately test the software.

The unified process

This *unified process* is also known as the *Rational Unified Process* (RUP), after the Rational Corporation who helped in the model's development. The Rational Corporation also develops CASE tools to support the use of the model.

The unified process is a unification of the various early object-oriented analysis and design models proposed in the 80s and 90s. It is an attempt to combine the best features of these various models which initially resulted in the *unified modelling language* (UML). The UML has become the standard diagrammatic language for modelling object-oriented software.

While the UML provides a modelling framework for developing object-oriented software, it does not provide any process model. This led to the development of the unified process, which is a process model for developing object-oriented software, and uses the UML as its modelling language.

The unified process is an incremental software process that is architecture driven, focuses on mitigating risk, and drives development through using use cases. Being architecture-driven, early iterations focus on building the portions of the software that will define the software's overall architecture. Focusing on risk, early iterations also focus on developing the high-risk portions of the software. Software

development iterations moves through five phases: inception, elaboration, construction, transition and production. These phases cannot be directly mapped on to the generic process framework activities: rather, each iteration contains some of the framework activities.

The *inception* phase is concerned with project feasibility: what should the software do, in broad terms rather than specifics, and what are the high risk areas? Should the development go ahead? Inception is usually a short phase, often having no more than one iteration. Little development usually occurs during the inception phase, but the software requirements are discovered using use cases (communication), and a small subset of these requirements (those with high risk, and which focus on the software architecture) are fleshed out (communication and planning).

Programming begins during the iterations of the *Elaboration* phase. Each iteration develops the requirements fleshed out in the previous iterations (modelling and construction), and chooses more requirements to flesh out (communication and planning) for development in the next iteration. The elaboration phase completes once all of the requirements have been fleshed out. However, this does not mean that communication and planning activities stop and do not occur in later phases: there is always constant communication with the customer and an understanding that requirements may change.

Much of the construction activity occurs in the iterations of the *construction* phase. While the iterations of the elaboration phase each had at least one meeting in which some use cases are fleshed-out and selected for development in the next iteration, all the use cases have already been fleshed out when the construction phase begins.

The *transition* phase contains the initial portions of the deployment activity: the software is given to the customer for evaluation (called *beta testing*, which we will discuss in Chapter 9, *Software Testing*). The customer's feedback will cause the software to be modified as required, and thus the transition phase includes some communication and construction activities. The *production* phase includes the final portion of the deployment activity: the software is now being used by the customer, and is monitored and supported by the software engineer.

The transition phase employs a technique called beta testing. Beta testing occurs when the software is given to the user to allow them to use the software and uncover any defects and deficiencies. There should be a formal communications framework for the customer to report their findings to the developers, and for the developers to assess these reports and to determine how to proceed.

Agile process models

Many of the process models we have just discussed have a perceived weakness: a lack of acknowledgement in the importance of managing change in the software life-cycle, and an over-emphasis on the process, tools, and documentation associated with them.

Agile process models were developed as a way to avoid these weaknesses. The *Manifesto for Agile Software Development* states that the core values of agile process models are:

- **Individuals and interactions** over process and tools.

- **Working software** over comprehensive documentation.

- **Customer collaboration** over contract negotiation.

- **Responding to change** over following a plan.

The motivation for this manifesto is that it is difficult to predict how software systems, the teams that develop them, and the context in which the software is used, evolve. The market conditions in which the customer wished to use the software could change, and the customer's needs will evolve to meet these new conditions, changing the software requirements. We may not even be able to decide on the requirements when the development work commences. Software engineers and their development methods must be *agile* enough to respond to all of these changes.

The customer is important in agile development. There must be effective communication between the customer and the developers in order to properly understand what it is that the customer needs.

The customer usually also works closely with the development team, allowing the developers to more fully understand their requirements, and allowing the customer to more fully understand the state of the software.

Apart from stressing closer communication between customer and developer, agile models also stress better communication between the members of the team creating the software. The most efficient form of communication is considered to be face-to-face communication, rather than documentation.

While agile development is strongly driven by the customer, it also recognises that any plans laid out to meet their requirements may change. This generally means that agile process models use an incremental / evolutionary approach to development, delivering multiple increments of the software to the customer. This allows the customer to have working software, to evaluate the software, and to ultimately allow the developers to more effectively respond to the customer's requirements.

Apart from not adequately dealing with change, prescriptive models also do not necessarily deal well with the differences between people.

For instance, people differ in the skills they have, and the levels in which they have these skills; they differ in how well they communicate, and in which mediums they communicate the best in (verbal or written, for example).

A process model may deal with the differences between people either with discipline (i.e., there are no options other than to follow the process activities as outlined by the process model) or with tolerance of these changes. While this is clearly a continuum, prescriptive process models can be characterised as choosing discipline over tolerance, and agile models as choosing tolerance over discipline. There is a trade off involved in shifting from a tolerant to a disciplined process model, and *vice versa*: while tolerant models are easier for developers to adapt to, and hence more easily sustainable, disciplined models are apt to be more productive.

The features of agile process models

The key features of an agile process model can be summarised as follows:

- The software itself is the important measure of the team's progress, rather than documentation.

- The development team has autonomy to determine how to structure itself, handle the development work, and apply the process model.

- Adaptability to change comes in large part through delivering software incrementally.

- Adaptability also comes from frequent delivery, so that customers can more easily examine the software and provide feedback.

- The process is tolerant: it is adapted to the development team's needs.

- Software is important, documentation less so: this means that design and construction are often heavily interleaved.

Extreme programming

We will discuss the most widely used agile process model, *extreme programming*. Extreme programming is an object-oriented development approach and provides four framework activities: planning, design, coding and testing.

Planning

Planning begins by creating *user stories*, which are similar to *use cases*, which we will cover in depth in Chapter 3, *Requirements Engineering*. User stories relate how the software will be used, and what functionality it will provide. The customer then prioritises these stories. The development team, in turn, determines the amount of development time required to develop the story.

The stories are grouped to form deliverables. These are the deliverables that will be given to the customer at each increment. Each deliverable is given a delivery date.

Importantly, the *project velocity* is determined at the end of the first increment: this is essentially the time take to develop the number of stories that were delivered in the first increment. This can be used to better estimate the delivery times for the remaining increments.

Note

New stories can be added at any time. The new stories are prioritised, and then added to an appropriate deliverable based on this priority.

Design

The design activity in the extreme programming process focuses around *class-responsibility-collaborator* (CRC) cards (see Chapter 5, *Object-oriented Analysis and Design*), which the developers use to organise the classes that need to be implemented in the software. These cards are the only design documentation produced using this process model.

When the appropriate design is difficult to decide upon, a prototyping method is employed. The design is quickly prototyped to evaluate the risks associated with it, and to estimate the required time needed to implement the story. This prototyping is called a *spike-solution*.

Extreme programming encourages developers to reorganise the internal structure of the code (without altering the program's behaviour) in order to make developing the software easier, and to make it less likely that bugs will be introduced in future work. This process of reorganisation is called *refactoring*.

Note

Because of the refactoring, the CRC cards often need to be modified. The design should not be viewed as something set in stone, but as something flexible, that changes as the software does.

Coding

Extreme programming has a number of distinctive coding practices. First, before coding begins, the process model recommends developing unit test cases to test each of the stories being developed in that increment's release. The developers then code towards satisfying those unit tests. This also allows the programmer to better understand how the code will be used and how the code should be implemented.

The second distinctive feature is *pair programming*: all code is written by pairs of programmers, with one developer programming and the other ensuring that the code follows an appropriate coding standard.

Testing

As just mentioned, unit tests are written before coding begins. It is also recommended that a framework is in place to run all the unit tests, allowing them to be easily and repeatedly run. This also allows for software testing to occur on a daily basis.

Apart from unit tests, *acceptance tests* are tests defined by the customer. They focus, by their nature, on the functionality visible to the customer, and will ultimately derive themselves from the stories used to develop the software.

Advantages and disadvantages

Extreme programming has the following advantages:

• Extreme programming is an incremental process model, and so the customer will have working software very early.

- The customer works closely with the developers, so the developers have a better understanding of the software requirements.

- Pair programming allows for quality checks of code as programming happens.

- Extreme programming has a strong focus on accepting changing project requirements.

Extreme programming has the following disadvantages:

- Much time might be spend re-coding the software, rather than focusing initially on a better design.

Differences in life cycle models and inconsistent use of terminology

There are no "officially" agreed set of terms used in descriptions of software development. Likewise, there is no standard set of stages in the waterfall-model (and many other models). When reading texts from difference sources, and when speaking with information systems professionals, it is wise to bear in mind the meanings of the terms being used, and if necessary clarify how the terms are being used.

Such differences are not a bad thing — they are a result of both the relatively small time that information systems and software development has been studied for, and reflect the dynamic viewpoints and interpretations of what we have learnt about software development to date.

Note

Do not take for granted that a particular book your are reading or person you are speaking to has the exact same understanding of the waterfall model, or of a term such as, say, "system design", that you have in mind.

Avoid having a single "correct" version of a life cycle model, stage or deliverable, especially since the models and deliverables should be adapted to fit the needs of the development team and project.

A useful learning activity to apply to each different reference source you use is to create a summary of the terms used and the model stages. As you work with different sources note the differences between the sources. Your own understanding will improve as you attempt to build an inclusive and general picture of the stages and deliverables of different views of the software life cycle.

Computer Aided Software Engineering: CASE

The role of tools in engineering systems development

Each activity in a process model has tasks to be performed and deliverables to be produced. Various software tools have been developed to support software developers using a particular process model. These tools support project management and monitoring and the use of any special techniques required by a particular process model, such as the UML. CASE (Computer-Aided Software Engineering) tools can also support the integration of several different deliverables to provide a consistent, overall model of the system being developed. Such a tool facility is usually said to be performing consistency or integrity maintenance and checking. Although there are simple, single-technique tools (such as for the UML), sophisticated tools exist to provide integrated support for many parts of a process model.

CASE tools

There are a wide range of software tools being used to support information system development. Some examples include:

- programming language compilers and debuggers

- project planning and costing applications

- help-file author applications

- version control support systems

- program code generators and "intelligent" program editors

- fully integrated suites supporting construction and consistency checking between multiple modelling techniques

The most sophisticated of CASE tools is the last in the list — the fully integrated tools supporting multiple modelling techniques. Some authors argue that any software application that helps in the system development process is a CASE tool, including:

- word processors

- spreadsheets

- general purpose drawing and painting applications.

We shall define a CASE tool as follows: a **CASE tool** is a software application (or integrated suite of application) whose sole purpose is to support one or more aspects of the software development process. The inputs and outputs of a CASE tool are either final system components, or artifacts whose purpose in some way supports system development and management.

The above definition excludes general purpose applications such as word processors and drawing packages.

It used to be common to make a distinction between three classes of CASE tool:

- Upper-CASE

- Lower-CASE

- I-CASE

Upper-CASE

"Upper-CASE" refers to CASE tools that support the early, abstract, "higher-level" activities of the system development process activities such as requirements elicitation, system analysis, creating logical design models, and so on. Activities which actually involve detailed "lower-level" design and implementation issues such as coding and testing are covered by "Lower-CASE" tools.

Example of Upper-CASE tools are the SELECT-Enterprise CASE tool, and ArgoUML. They are sophisticated tools, supporting activities such as logical data modelling and data flow process modelling techniques.

The following screen shows the Entity-Relationship Diagram window of SELECT-Enterprise:

Figure 2.7. SELECT-Enterprise screenshot
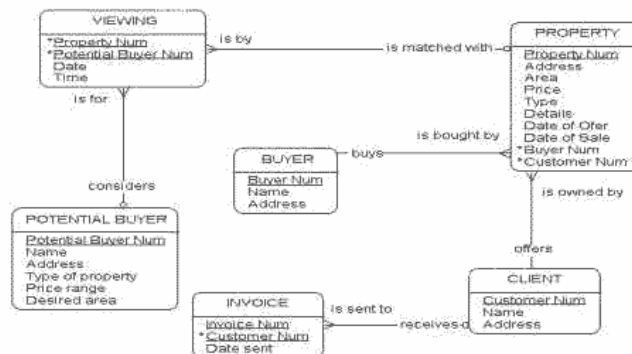
These notes will, however, mostly be making use of UML CASE tools, such as ArgoUML and Umbrello.

Lower-CASE

Activities which involve "lower-level" design and implementation issues are known as "Lower-CASE". Examples of Lower-CASE tools are program code compilers and debuggers, interface building programs and so on.

The following is a screen shot from a Lower-CASE tool — Symantec's VisualCafe environment for Java software development. In this screen we can see some Java program code and the corresponding interface objects on screen:

Figure 2.8. Java code
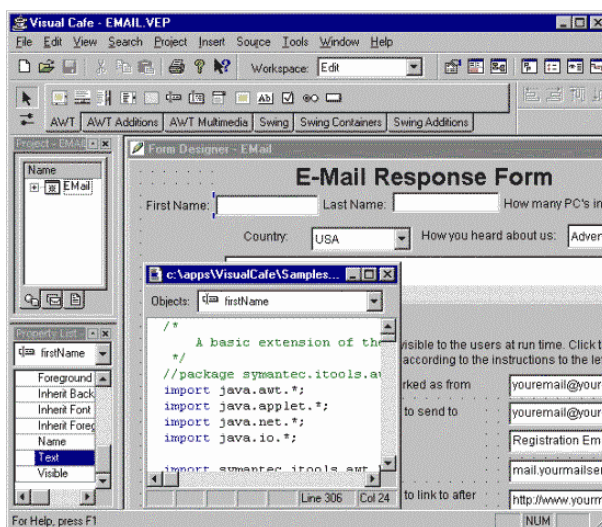
The following screenshot is another example of Lower-CASE — the command-line based java interpreter provided by the *Oracle Corporation* (previously owned by *Sun Microsystems*):

Figure 2.9. A command-line Java interpreter
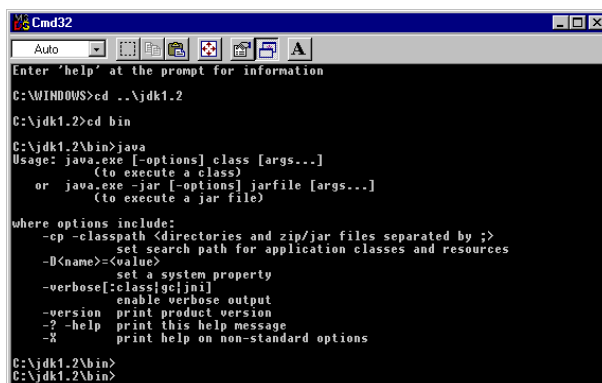
I-CASE

The "I" in I-CASE stands for "Integrated". The most powerful CASE tools offer functions to integrate a range of different kinds of models for a system, to manage a more complete picture of a system. Such integrated tools can check multiple models for consistency, in addition to assisting system modellers to handle the "cascade" of changes that occur when a change in one model requires a related change in another.

Each I-CASE tool tends to be developed to support a particular process model. Those CASE tools that do not provide integration between multiple models or products may be used for many process models — for example, regardless of the model, at some point a code compiler (or interpreter) will be needed.

IPSE — integrated project support environment

Integrated Project Support Environments are integrated applications to support project management. These usually provide some combination of project planning/scheduling, costing, version control of models and software components, and support for project management documentation and reporting.

Critique of CASE

Advantages of CASE

CASE offers a number of advantages for the development of systems:

- CASE tools allow for the faster development of models, and provide support for creating diagrams.

- Some CASE tools offer simulation of the system being modelled (e.g., given an event, possible system responses can be traced).

- CASE tools can be used to maintain a central, system dictionary, from which all models draw their components.

- CASE tools can help the team be more positive about improving models when errors or inconsistencies are identified.

- CASE tools provide consistency checking between different types of model.

- CASE tools can provide notation and documentation standards in and between projects.

- CASE tools provide navigation support between related parts of models and diagrams.

- They provide automated documentation and report generation from the system models and main dictionary.

- IPSE CASE tools increase the rigour of project planning and management, and support straightforward re-planning and response to unexpected events.

Disadvantages of CASE

Although there are clearly many advantages, there are a number of potential problems with CASE:

- They can be very expensive.

- Project staff require expertise (or training) on the software.

- A particular CASE tool based around a single methodology will force a project to commit more strongly to the methodology than they might wish to have done.

- There is a danger that high quality models and diagrams can lead to poor software development — the models may look impressive, but it is the quality of the *system* and not the diagrams that is fundamentally important.

Remember that having consistent models of the software does not guarantee that the software will meet the user requirements.

Conclusion

It would be a poor project management these days not to take advantage of one or more CASE tools for all but the very smallest of system development projects. Software support for the activities and

production of deliverables during system development has progressed with the profession's increased understanding and modelling of the system development process.

Some I-CASE tools provide support for a wide range of high- and low- level products of the life cycle, thus blurring the distinction between Upper- and Lower- CASE — such products can work from abstract models such as those produced by the UML, and provide skeletal databases and computer programs.

Review

Questions

Review Question 1

What is a life cycle model?

A discussion of this question can be found at the end of this chapter.

Review Question 2

What are the activities of the generic software process?

A discussion of this question can be found at the end of this chapter.

Review Question 3

What features of the waterfall life cycle model separate it from other models?

A discussion of this question can be found at the end of this chapter.

Review Question 4

Describe the disadvantages of the waterfall model.

Answer to this question can be found at the end of this chapter.

Review Question 5

What are the main features of the prototyping life cycle model?

A discussion of this question can be found at the end of this chapter.

Review Question 6

What are the points of the *Agile Manifesto*?

A discussion of this question can be found at the end of this chapter.

Review Question 7

What is a CASE tool? Provide some examples of CASE tools. Try to also provide examples of tools that you may have used before.

A discussion of this question can be found at the end of this chapter.

Review Question 8

Describe some of the advantages of CASE?

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

A life cycle model, or *software process model*, is a description of the best practices for engineering and developing software. It is usually broken down into stages, describing the deliverables produced during each stage, and describing the order or iterative cycles of the stages, and when each stage would be appropriate to perform.

Discussion of Review Question 2

There are five generic activities:

- Communication

- Planning

- Modeling

- Construction

- Deployment

Discussion of Review Question 3

The waterfall model breaks the system development process into a linear sequence of stages, each involving a specific activity. Each stage produces deliverables that become the inputs to the following stage. Once a stage has been completed it cannot be revisited — so decisions made in early stage are committed to and determine what happens at later stages.

Discussion of Review Question 4

The Waterfall model assumes that there is a full understanding and specification of the requirements at the beginning of the project, and that these requirements will not change during development. Since the deliverables at each stage is usually not software, it is easy for none-software related deliverables — and the bureaucracy surrounding these deliverables — to become the focus of the software development. Because of the linear nature of the model, problems identified in earlier stages become progressively more difficult and expensive to fix in later stages.

Discussion of Review Question 5

The prototyping model attempts to produce a small version of the final system that has only *some* of the final software's functionality. Prototype life cycle models allow for the developers and users to explore the uncertain requirements of the software system, and an analysis of the prototype can form the basis of a requirements specification for further development. Because prototypes are generated quickly and are incomplete, once the requirements are better understand the prototype is discarded and development restarted using another process model.

Discussion of Review Question 6

There are four points:

- **Individuals and interactions** over process and tools.

- **Working software** over comprehensive documentation.

- **Customer collaboration** over contract negotiation.

- **Responding to change** over following a plan.

Discussion of Review Question 7

CASE tools are software whose sole purpose is to support one or more aspects of software development.

Common examples include compilers and debuggers — if you are using Ubuntu or OS X, that will often be *GCC* for the compiler, and *GDB* for the debugger. There are also many CASE tools for planning and software costing, such as *Microsoft Project*. Other case tools include those for authoring help files, version control systems (such as *Subversion*, *Bazaar*, *Mercurial* and *Git*), code generators and program editors (such as *Emacs*, *Visual Studio* and *XCode*), and tools providing access to modelling techniques such as the UML (for example, *ArgoUML*).

Discussion of Review Question 8

Case tools allow for the faster development of various software models, and can sometimes even simulate portions of the software from these models. Some tools allow a central dictionary to be maintained where each model can take their components from. They can automatically check for consistency between various models, can provide a standard for notation and documentation, and can provide easy navigation between different portions of the models. They can support planning the activities of the software development life cycle, and can support the programmers directly with their development (by providing access to documentation, report generation, by keeping track of changes made to the source code, and so on).

Chapter 3. Requirements Engineering

Objectives

At the end of this chapter you should be able to:

- Explain the need for requirements engineering.

- Give a series of steps for use in performing requirements engineering.

- Create and interpret use case models.

Requirements engineering

The first two activities in the generic process framework is that of *communication* and *modelling*. A large portion of these activities are concerned with discovering the requirements of the software which the customer is asking to have developed. This chapter deals with this process of *requirements engineering*.

Note

As with all other activities in a process model, requirements engineering should be tailored to fit the developers creating the software, the product being created, and the overall process model being employed. In the previous chapter you could already see this happening with the *extreme programming* software process model, which limits modelling to the creation of CRC cards (see Chapter 5, *Object-oriented Analysis and Design*).

Requirements engineering is concerned with understanding the software system that the customer has requested. It provides the base on which software design and programming can proceed. Importantly, if the developers do not adequately understand the requirements, it is very likely that the software will not meet the customer's needs. This makes understanding the customer's requirements important to the success of the development project.

What is requirements engineering?

At its most essential, requirements engineering is focused on discovering *what* it is that should be developed (and not *how* it should be developed). There are a number of aspects to this:

- What does the customer want?

- What does the user require in order to use the system?

- What will the software's impact on the users be?

To discover this information, requirements engineering contains a number of overlapping steps:

1. **Inception**, in which the nature and scope of the system is defined.

2. **Elicitation**, in which the requirements for the software are initially gathered.

3. **Elaboration**, in which the gathered requirements are refined.

4. **Negotiation**, in which the priorities of each requirement is determined, the essential requirements are noted, and, importantly, conflicts between the requirements are resolved.

5. **Specification**, in which the requirements are gathered into a single product, being the result of the requirements engineering.

6. **Validation**, in which the quality of the requirements (i.e., are they unambiguous, consistent, complete, etc.), and the developer's interpretation of them, are assessed.

7. **Management**, in which the changes that the requirements must undergo during the project's lifetime are managed.

Requirements engineering will usually result in one or more *work products* being produced. These products, taken together, represent the software's *specification* (see the specification step previously mentioned, and detailed below). These work products, however, do not have to be formal, written documents — indeed, the work products can be a set of models, a formal mathematical specification, a collection of use cases or user stories, or even a software prototype.

Note

These steps are *overlapping* for a variety of reasons. You should be able to notice that some of them, such as management, must occur throughout the *communication* and *modelling* activities. Negotiation will also occur at each of the various requirements engineering steps. More importantly, a process model which understands that requirements may be (initially) poorly understood, and that they may change through the project's lifetime, will also iteratively collect and detail the use cases (consider the unified process model from the previous chapter). This iterative process will forcefully overlap all of these steps.

The steps in detail

Inception

The requirements engineering process begins by examining the problem which the software should solve and gaining an understanding of both the problem's nature and the nature of the desired solution. This should be done in a context-free manner, that is, a manner which does not presume to know anything concerning the problem, the customers, the users, and the requested solution. The following questions may be asked:

• Who is requesting the software?

• Who will use the software?

• What is the benefit that the software will bring?

It is important to identify the **stakeholders** in the project. Stakeholders are the people who will find benefit in the project and the software being developed. They may include:

• Customers

• End users

• Business operations managers

• Product managers

• Advertising and marketing staff

• Software engineers

• Support engineers

Each of the stakeholders will have a different view on what the software product should do and on what the software engineers should focus on. This might be on creating "sexy" features (from the marketing department), staying within budgets and deadlines (from managers), maintainability (from

support engineers) and so on. Out of these views, the requirements engineer should determine which requirements there are a consensus on, and on which requirements the stakeholders disagree. Resolving disagreements between stakeholders makes up the negotiation step.

Something to consider during inception is the effectiveness of the communication between the requirements engineer and the customers. This may be done by, for example, asking the customer if they feel that they have been asked appropriate questions concerning their problems, and if the person communicating with the requirements engineer feels that they are (or are not) the person who should be answering the engineer's questions.

Elicitation

This step is concerned with identifying the overall problem the software is attempting to solve, proposing solutions, negotiating between the differing approaches to solving the problem, and finally specifying a basic set of requirements.

This can be done by calling a meeting between all of the stakeholders. It is important to nominate someone to act as a *facilitator*, who will guide the meeting. Each of the attendees should bring to the meeting a list of:

- objects that make up the system's operating environment

- objects used by the system (such as those things which make up the input to the system)

- objects produced by the system

- the services that interact with these objects

- various constraints, such as time and budget constraints, interoperability constraints, performance restraints, usability constraints, and so on.

Elaboration

This step involves expanding on the requirements defined in the previous two steps, and from these requirements producing an *analysis model*, which is a technical model of the software and its functions.

The construction of analysis models will be discussed in detail in the following two chapters.

Negotiation

This step involves negotiating between the various stakeholders in order to remove any conflict in the requirements. A useful technique for resolving these conflicting requirements is to provide each of the stakeholders with a finite number of *priority points*. They may then allocate points between the conflicting requirements as they see fit. The overall importance of any requirement can then be determined by the number of priority points that it has received.

Specification

The specification step produces the final product of the requirements engineering process. It describes the software, both its functions and constraints. The specification need not be a written document, but could also be a graphical model (such as those produced using the UML), software prototype or formal model, or a collection of these.

Validation

This step is concerned with ensuring that the gathered requirements in the software specification meet certain standards of quality. For example, have the requirements been written to the proper

level of abstraction, or do they provide too much technical detail for the given stage of development? Is the requirement necessary, or something not essential to the software? Are the requirements unambiguous? Do requirements contradict other requirements?

A useful action during validation is to ensure that each requirement has a source attributed to it. In this way, if more information is required, the requirements engineers know who to contact.

Management

Requirements change over time; requirements management is concerned with controlling and tracking change in the requirements.

Requirements management proceeds by associating requirements with various aspects of the software engineering process. As these aspects are changed, the requirements associated with them can be easily identified and changed. As these requirements are changed, all aspects of development associated with the modified requirements can be examined, and in this way the changes can more easily be propagated through the project.

Such an association between requirements and aspects of the project can be done using a table: each row in the table represents a specific requirement, each column an aspect of the software project. The entries mark whether a requirement is associated with that aspect.

Use case modeling

Use case modelling is a useful tool for requirements elicitation. It provides a graphical representation of the software system's requirements.

The key elements in a use case model are **actors** (external entities), and the **use cases** themselves. In outline, a use case is a unit of functionality (a requirement), or a service, in the system. A use case is not a process, or program, or function.

Because use case models are simple both in concept and appearance, it is relatively easy to discuss the correctness of a use case model with a non-technical person (such as a customer).

Use case modeling effectively became a practicable analysis technique with the publication of Ivar Jacobson's (1991) book "Object-oriented software engineering: a use case driven approach". Jacobson has continued to promote this approach to system analysis to the present day, and it has now been formalised as part of the UML. However, use case modeling is not very different in its purpose and strategy from earlier techniques, such as structured viewpoint analysis.

Use case modeling in the UML specification

The *Unified Modeling Language* (UML) represents a deliberate attempt to standardise the modeling notation used in software engineering, particularly object-oriented development. The widespread uptake of the UML is a result largely of two factors. First, it is driven by some of the most influential proponents of object-oriented development, including James Rumbaugh, Grady Booch, and Ivar Jacobson. Second, it has broad support from major business concerns in the software industry, including Microsoft, IBM, Hewlett-Packard and Oracle.

The notation specified for use case modeling by the UML is not very different from that originally proposed by Jacobson, so early books and articles on use case modeling that follow the Jacobson strategy are still useful reading.

It is only fair to point out that not all experts support the UML effort, and it comes under regular and harsh criticism, some of it fair. For example, one criticism is that there is not good enough integration between the different components of the UML (e.g., between use case and class modeling). No doubt this will improve in time. In due course you will be able to make your own judgement on this issue, but
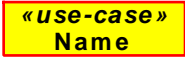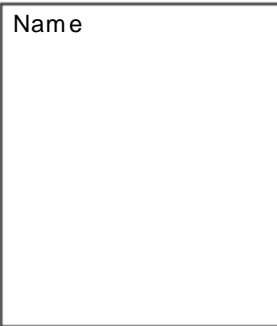
is important to keep sight of the fact that the UML is an international standard for software modeling, and any software professional needs to understand it.

The UML is under continuous development, and at the time of writing the latest version is 2.2. The definitive reference for the UML notation is the UML specification, which is available from the Object Management Group's Web site [http://www.omg.org/technology/documents/formal/uml.htm]. However, while this is an authoritative document about the UML, it is not a good document from which to learn about the UML.

It is important to understand that the UML is a specification for a modeling language. It is most emphatically not a software design methodology. Although the UML states the symbols that are to be used in use case modeling, and how they are to be interpreted, it does not say when, or even if, use case modeling should be applied. We shall have more to say about this later.

Use case modeling symbols

This section presents an overview of the symbols used in use case modeling; the important ones will all be discussed in detail later.

Symbol	Name	Interpretation
or	Actor	An entity (human or otherwise) external to the system, and which interacts with it
or	Use case	A service or unit of functionality
	System boundary	Indicates the division between the system being designed and the rest of the world
	Communication association	The line indicates that a particular use case is associated with a particular actor. The name is optional and often omitted. An arrow can also optionally be used; where present it does *not* indicate a flow of information (such as in a data flow diagram)
	Use case association	Indicates that to use cases are related in a particular way, e.g.,

Symbol	Name	Interpretation
		the one use case's behaviour includes the behaviour of another use case
«name»	Stereotype	Indicates that the symbol it is attached to belongs to a particular category
	Generalisation	Indicates that the two symbols it connects are related by a generalisation-specialisation relationship. For example, one actor is a sub-type of another, or one use case is a type of another. Both the name and the stereotype are option
	Note	The designer may, and should, qualify any part of the model with a textual note if it improves the clarity of the design

Note

The above images were created using the Umbrello Software package [http://uml.sourceforge.net].

Actors

An actor is any entity, human or otherwise, that is external to the system being designed. Two symbols are available in the UML specification:

Figure 3.1. Actor representations

Name
An actor can be represented by a stick figure.

«actor»
Name

Alternatively, an actor may be represented by a class with the «actor» stereotype.

The "stick figure" symbol is the more expressive, but can lead to confusion if the actor is not in fact a human but a machine. The rectangle symbol is the standard UML symbol for a class. What this symbol says is that the entity is a class that is a member of the category "use case". The reason why an actor is a type of class will be discussed later. Because of its greater expressive power (that is, an ability to make a more immediate impression on the viewer) it is probably best to use the stick person figure where possible.

Since use cases are technically classes, and by convention class names start with a capital letter, names of use cases should also begin with a capital letter. The UML specification does not insist on this, but it is common practice.

Two interesting philosophical issues are associated with actors. First, should the actor be a person, or the part of the computer with which they interact? For example would we ever want an actor called "Keyboard" or "Printer"? In designing a word processor, for instance, there is only one human

user and they fulfil all the roles within the system. But you may wish to distinguish between, say keyboard, mouse, and printer actions. However, distinguishing between different pieces of hardware is probably inappropriate at this level, and it could be argued anyway that use case modelling is not a helpful way to begin the design of a word processor. On the other hand, a supermarket stock control system may accept input from a bar-code scanner at the checkout, when the cashier registers the prices of the customer's purchases. Is the actor here the cashier, or the bar-code scanner? Perhaps neither is appropriate, as the cashier can probably enter the identities of products manually if the bar-code scanner does not work. Do we want to distinguish between different techniques for entering item details at the highest level of analysis? Probably not. In this case we could perhaps invent a more abstract actor (called, perhaps, "point-of-sale") that provides the key message: that something at the point-of-sale interacts with the system whenever a purchase is made. In summary, then, when a human being interacts with a computer system, it is not necessarily the case that the human is identified as the actor. It is often clearer to use a non-specific entity as the actor.

The second philosophical issue is concerned with whether an actor has to be an active participant in an interaction. For example, in a computer-based building security system, do we want "Burglar" as an actor? A burglar does not actively interact with the system; indeed they would probably rather not interact with it at all. Moreover, a burglar may interact with the system in a number of different ways, all passively. In this case, it may well be better to identify the sensor devices as actors. What about the fire alarm? Presumably "Fire Detector" is a better actor than "Fire"?

The UML gives us no guidance on these issues; although the letter of the UML specification is that actors can be any external entity, the spirit of the standard seems to be that actors are people who interact with the system in a rich, complex way. There is little concern for "trivial" actors like printers and fire detectors.

These complications notwithstanding, actors do not have to be human beings; they may be external computer systems. For example, we may be designing a system for allocating staff work timetables that draws information from a central record of employees which is part of the payroll system. In this case, "Payroll System" would be an actor.

You should bear in mind that you will never implement an actor; by definition an actor is external to your system. You will, however, implement the interfaces used by actors to interact with the system.

Actors as roles

A decision that the use case modeller has to make is whether to treat (human) actors as expressions of a person, or as expressions of a role within an organisation. Perhaps an example will help to clarify this issue. Suppose that you are designing a computer system to automate the operations of a large library. The system should maintain the library catalogue, provide information to staff and readers, check books in and out, provide guidance on re-shelving returned books, manage inter-library loans, warn users about overdue books, and manage collection of fines and subscriptions.

Most people faced with this problem begin by considering the types of people who will use the system. Two such groups come to mind immediately: library users and library staff. In a library one will often find that staff members have very general job descriptions, and will take a hand in most of the normal operations of the library. We will refer to these people as "librarians". So far our use case model has two actors: Librarian and User, and these two actors interact with all the use cases that we could identify. In this case we are treating "Librarian" as an expression of a person. We could get a lot of information about what services the computer system should provide by reading the job description of a librarian.

The problem with this approach is that it has no high-level structure, and is therefore not very expressive. We could convey much the same information as the use case model by merely writing a list of library services. There is no scope for simplification and management of the model by generalisation (see later). Furthermore, it carries the implication that librarians are interchangeable, and any librarian can do the work of any other. Even if this is true at the time the system is designed, it may not continue to be true. For example, more junior staff members may not have authority within the system to order

new books for the library. A person managing the stock of books is interacting with the system in a totally different way to the person who is, for example, signing up new library users. The fact that these to job functions may perhaps be carried out by the same physical human being is irrelevant; they are totally different roles within the system, and should be considered to be different actors.

Of course, we could now go to the opposite extreme and create a new actor for every service the system provides, but this leap from the frying pan into the fire would still result in a model with no structure that is difficult to simplify. In addition, there may be a loss of expressiveness. For example, in the library system it is quite likely that both staff and users can browse the catalogue of books. They will quite possibly do this in an identical way, and see exactly the same information. Moreover, this activity is completely separate from any other functionality the system may provide. So we may be tempted to create a new actor (perhaps called "Catalogue Browser") to model this role within the system. But we will have lost a very important piece of information by doing this: the fact that both staff and users can browse the catalogue implies that we have to put computer terminals or workstations on both sides of the counter, so to speak.

So what is the correct approach? In short it is the one which leads to the simplest correct model which is still adequately expressive. Of course it is more important that the model is correct than that it is simple. To arrive at this point may require that your choice of actors be modified several times during the course of construction of the use case model.

Identifying actors

In the last section we considered the distinction between actors as people and actors as roles, and clearly this is an issue that needs to be taken seriously by the requirements engineer. However, it only becomes an issue when we know enough about the way the system is to be used to have a potential set of actors to hand. We now have to consider the situation where the analyst knows nothing at all about the system to be developed, and doesn't have the first idea what the actors are. Many development jobs begin like this.

The analyst's first recourse is to the stakeholders identified through the earlier requirements engineering steps. The stakeholders should certainly be on the initial list of actors.

You may then ask what information is to be manipulated by the system, and where that information will come from. All information that is not present in the system at the instant it begins work will be coming from an actor of some sort. If there is not enough information to do this, then this is a sign that further information must be elicited from the customers.

Other potential actors include all the computer systems with which the system will interact, and any other hardware devices (including hardware such as printers, bar-code readers, and so on).

There are also various "standard actors" that all systems of any complexity are going to need, and which may not have been considered by the clients. These include the person (or rather the role) which maintains the system after it has been put into service, the person who performs software upgrades and tests, the person who carries out and checks backups (if not automatic), and so on.

When developing a software system to replace a manual procedure, or an older software system, actors may be identified by watching people go about their daily work, and by speaking to people who are experts on the business the system has to carry out.

It is probably not possible to identify actors without some consideration of the services that the system provides. In practice design work will switch between consideration of the actors and consideration of the use cases.

Individual actors and classes of actors

All of the forgoing was, we hope, largely common sense. Now it is time discuss a technicality. We will state the principle, and then go on to explain it.

The principle is this: the UML specifies that an actor in a use case is a class, and individuals are instances of that class.

The actors in a use case model do not represent individuals (individual humans or individual computers), but classes of individuals. For example, an actor called "Customer" models all those properties that customers have in common; it is, in effect, the class of all customers. It will be assumed by anybody who reads your model that the ways in which the Customer actor interacts with your system will apply to all customers. So if some customer or group of customers is expected to behave differently, then we need a different actor to handle this case.

This may seem trivially obvious, but the complication is that the above principle is true even if there is only one instance of an actor. For example, a company may have only one managing director, and may be constrained by law only to have one, but if the managing director interacts with the system you are designing in a specific way, then they are a class, not an individual. The problem is that although the designer understands intellectually that actors are classes, they may still have a specific individual in mind, leading to a bad model. The reason is this: suppose Joe Bloggs is a bank clerk; there can be different types of bank clerk, but it is meaningless to talk of different types of Joe Bloggs. And the ability to simplify a model by identifying where one thing is a type of another thing is a key feature of use case modeling, and indeed of all types of object-oriented design.

System boundary

The system boundary demarcates the system being designed from the rest of the world. It is denoted simply by a box with the name of the system in the corner.

Figure 3.2. The system boundary

Name
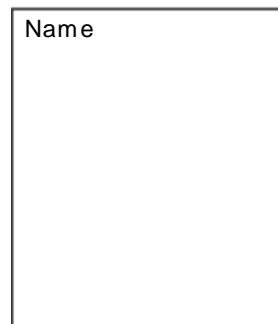
Use cases are inside the box, actors are outside. Because different symbols appear inside the box and outside, in practice it is usually unnecessary to show the system boundary explicitly. One circumstance in which it is necessary to show it is if you wish to show the use case models of two or more different systems on the same diagram. This may be necessary if you are designing a system which interacts with another in a way that is too complex for you to show the external system as simply an actor.

Some modeling software will draw the system boundary automatically.

In the design of a large, complex system, it is said that the boundary may "move" during the design operation. In fact what is happening is that decisions are being made about what functionality is the responsibility of the current design exercise, and what is not. For example, it may turn out that during the analysis of a staff payroll system, management of staff health records — which was previously the responsibility of another system — is seen to be more appropriately part of the payroll system. In effect, the system boundary has "moved" to encompass part of another system. Needless to say, such movement needs to be settled as early in the development process as possible.

Use cases

In the UML, a use case can be represented in two different ways:

Figure 3.3. Representations of use cases

Use cases may be represented by a title within an ellipse.
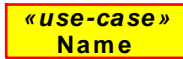
Use cases may be represented by a class with the «use case» stereotype.

Most designers will immediately recognise the oval as a use case (the oval symbol has no other meaning in the UML) so this symbol is to be preferred where available. The other symbol reflects the fact use cases are technically classes (just as actors are), whose category is "use case". We will discuss later what is meant by a class of use cases.

Although use cases are central to use case modeling — and indeed to many object-oriented development strategies — there is surprisingly little general agreement on what a use case is. In fact, there are people currently working on doctoral theses whose subject is what a use case is supposed to be showing. The following definition is taken from version 0.8 of the UML specification:

> A use case is a generic description of an entire transaction involving several objects.
>
> —UML 0.8

This rather terse definition was all the UML had to say about the nature of use cases at that point. In version 1.3, there is the following, somewhat expanded, definition:

> The purpose of a use case is to define a piece of behaviour of an entity without revealing the internal structure of the entity. The entity specified in this way may be a system or any model element that contains behaviour, like a subsystem or a class, in a model of a system. Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again. A use case describes the interactions between the users and the entity as well as the responses performed by the entity, as these responses are perceived from the outside of the entity . A use case also includes possible variants of this sequence, e.g. alternative sequences, exceptional behaviour, error handling etc. The complete set of use cases specifies all different ways to use the entity, i.e. all behaviour of the entity is expressed by its use cases.
>
> —UML 1.3

Versions 2.2 states:

> A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.
>
> —UML 2.2

In fact, these are very general descriptions, and not particularly helpful to the practitioner; here are some more pragmatic views of the use case.

Use cases as services

In this view, each use case provides a particular service to the users of the system. Examples of services may include withdrawing money from an account, printing a report, reserving a theatre ticket, and ordering a book. Every time an actor interacts with a use case a transaction occurs; this transaction takes time and may have a number of alternative paths.

Use cases as business processes

This view of use cases seems to be favoured by people with an interest in business process modeling. In this view, a use case is an activity of the business, such as ordering stock, issuing a check to clear an account, or checking a customer's credit-worthiness. Superficially this view is not that different from the view of use cases as services, but the implication here is that use cases are best characterised by a discrete series of actions, perhaps illustrated by a flowchart (the UML equivalent of a flowchart is an activity diagram; expanding use cases into activity diagrams is quite common practice).

Use cases as increments of functionality

In this view, adding a use case to the system is equivalent to adding some extra functionality, with the proviso that the new functionality be largely self-contained.

These views are not mutually exclusive, and all are compatible with the UML definition. However, the view that the designer adopts will subtly influence the character of the model produced.

Whichever view of you take of use cases, it is important to remember that a use case is a thing, not an event or a process. It is not uncommon for novices to generate use cases with names like "print" or "mouse moved". "Print" is a process or action. It may be a valid use case, if you mean by "print", "the facility to allow a user to obtain a printed output". Some experts recommend that you force the names of use cases to be nouns to reinforce this meaning. In this case "Print Service" or "Print Facility" might be better names. "moused moved" is an event, a thing that happens to the system.

Relationships between use cases

It is possible, and usual, to show that one use case is related to another. The standard notation for this is a dashed line:

Figure 3.4. Use case association

An association between use cases is typically shown with a dotted line.

A stereotype can also, optionally, be given to the relationship. This indicates the type of relationship, of which the most important are «include» and «extend». An example of an «include» relationship is show below.

Figure 3.5. Use of stereotypes in use case relationships
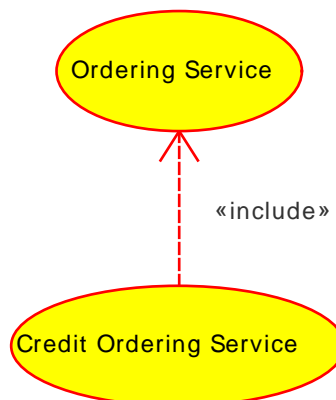
In this example, "Credit Ordering Service" includes "Ordering Service", that is, all the behaviour of "Ordering Service" will be invoked whenever "Credit Ordering Service" is invoked. This is sensible if ordering something on credit is the same as ordering something with payment, plus a bit extra.

Typography

Stereotypes in the UML are enclosed in guillemets: « and ». Often, if the typist or application is unable to represent these symbols, << and >> (two less-than or greater-than signs) are used instead. However, now that Unicode display is available on most operating systems, the guillemets are to be preferred. The Unicode code points for « is u+00AB, and » is u+00BB. Your operating system will no doubt have an easier input method available to you than entering Unicode code points directly.

In the UML, «extend» is similar to «include», with the distinction that with «extend» the behaviour of the extending use case will not always be invoked.

A note on terminology and semantics

The UML uses the term «extend» here because Jacobson used this term in his earlier work on use case modeling. Unfortunately, many authors use the term "extend" to mean a generalisation relationship — in other words, that one thing is a type of another thing. This is particularly relevant in Java programming, where the word "extend" has this (generalisation) sense, and not the sense defined in the UML. To further complicate matters, the «include» relationship does denote a kind of generalisation: a "credit ordering service" is a type of "ordering service". These complications have little impact on use case modeling or on programming in practice, but they do cause confusion among students (and others), and it is good to be aware of them.

Describing and specifying use cases

A use case diagram on its own may well be a useful method for describing the large-scale structure of a system; however, it is of very limited use as the input to a more detailed design operation. In practice it is necessary to describe use cases in a more detailed fashion if the model is to be interpreted properly.

Such a description is not only an aid to communication, it is an aid to *validation*. **Validation** is the process of ensuring that a specification is correct, in contrast to **verification**, which is the process of ensuring the product meets its specification (see Chapter 9, *Software Testing*). If the analyst cannot describe a use case in a way that makes sense to someone else, then one of two things has happened:

- The analyst has made an error: the use case is not valid and should be removed and its functionality placed elsewhere;

- The analyst has insufficient information about the system being developed.

Either case needs to be corrected.

How do we describe use cases? Two approaches are in widespread use.

- Plain text. This is the approach recommended by the UML specification. Normally a few paragraphs of text should be adequate. Most experts recommend that the text used should be in language terms which the customer uses; that is, a system for controlling a steel mill should be written in the language of a steel producer, not a software engineer, even though the latter may allow greater technical accuracy. The reason for this is simple: only the clients can confirm that the use case model is correct, and that they meet the requirements of their application. While the analyst can be sure that the model is logically consistent, and can be implemented, this is not good enough grounds for proceeding with development.

- Using a sub-model. By this we mean using a further, more detailed model, to clarify the use case. Ultimately, as we shall see, a use case will be expanded into a *collaboration diagram*, that is, an interacting group of classes. However, this does not describe the use case in the sense meant above; this will probably not help customers to determine whether the model represents a system that meets their needs. A common choice of a sub-model is the *activity diagram*. This diagram is a relative

newcomer to the UML, and is not very different from a traditional flowchart. It shows the sequence of steps that are carried out when a use case/actor transaction occurs, and can show alternative sequences of operations which are selected according to some condition of the system. The use of activity diagrams is beyond the scope of this chapter, but should be explained in any UML textbook and, of course, in the UML specification.

Individual use cases and use case classes

Use cases are technically classes. Thus a use case does not represent a particular delivery of a service, or use of some functionality, but *all* conceivable deliveries of the service.

If a use case is a class, then the individuals / instances are the specific cases of a transaction occurring between the use case and one or more actors.

If all interactions between the actor(s) and the use case are identical, then the distinction between the use case class and its individual instances is not important to the analyst. It becomes important when a use case can behave in many different ways, that is, the individual instances of the use case class are different. If these individuals are important enough to be documented, they are called scenarios.

For example, suppose we are developing a computer system that allows people to place credit-card orders for our customers' products using a Web browser. We have identified a use case called "Place Order", which represents an ordering transaction between the customer and the system. This use case has very complex behaviour, because there are many things that can go wrong while placing the order. For example, the credit card company may not authorise the funds transfer, the credit card number may be invalid, the connection to the credit card company may be out of service, and so on. In all cases the use case must exhibit behaviour that tries to recover from these errors. For example, if the user enters an invalid card number, they must be given the opportunity to correct it and try again.

For the purpose of documenting the use case we may use a model like an activity diagram to show the general behaviour, but support it by describing a number of possible complete transactions, where different errors are encountered and corrected. These descriptions are the scenarios of the use case. Including scenarios for complex use cases helps the analyst to be sure that the use case is properly specified, because the clients will be able to understand the scenarios (which are in plain language), and the analyst and their colleagues will be able to check that the scenarios are compatible with the general pattern of behaviour specified for the use case.

The concept of generalisation

Generalisation is one of the most important concepts in use case modeling, and indeed in object orientation in general. When stated as a principle it appears trivially obvious, but it has profound implications. The principle is this:

Generalisation

Entity A is a **generalisation** of entities B, C... if the behaviour, attributes and associations of B, C... are found in A, and no properties of A are absent from B, C...

We say that entities B, C, and so on, *inherit* the properties of A. This can also be stated as: A is a generalisation of B and C if B and C are *types* of A (we could also say *sub-types* or *sub-classes*); A is a generalisation of B and C if B and C *extend* the functionality of A, while retaining all of A's functionality

Here is trivial example: Dog and Cat are types of Mammal, because all dogs and cats share the basic properties of mammals (e.g., having fur, being warm-blooded, having four limbs). We may say that Mammal is a generalisation of Dog and Cat; alternatively we could say Dog and Cat are specialisations of Mammal, or types of Mammal, or sub-classes of Mammal. Note that when we say that Mammal is a generalisation of Dog and Cat we are not saying anything about whether there are other types of Mammal; there may be, but we haven not specified any.

Another example: A business wishes to automate some of its sales procedures. Preliminary interviews reveal that there are a number of staff roles in the Sales department. A salesperson can place orders on behalf of customers and check the status of these orders. A technical salesperson has the same duties, but additionally is able to provide customers with detailed technical advice (which we would not expect an ordinary salesperson to be able to do). A sales supervisor is a salesperson, with the additional responsibility of creating new customer accounts and checking their credit-worthiness. It is reasonable to assume that Salesperson is a Generalisation of Technical Salesperson and Sales Supervisor, because the technical salesperson and sales supervisor have all the properties of a salesperson, and some extra.

We can construct an outline use case model to show these relationships. We will assume for the sake of simplicity that the use cases are "Place Order", "Check Order", "Create Account", "Check Credit", and "Technical Advice". Without generalisation, we obtain the following model:

Figure 3.6. A use case example, without generalisation
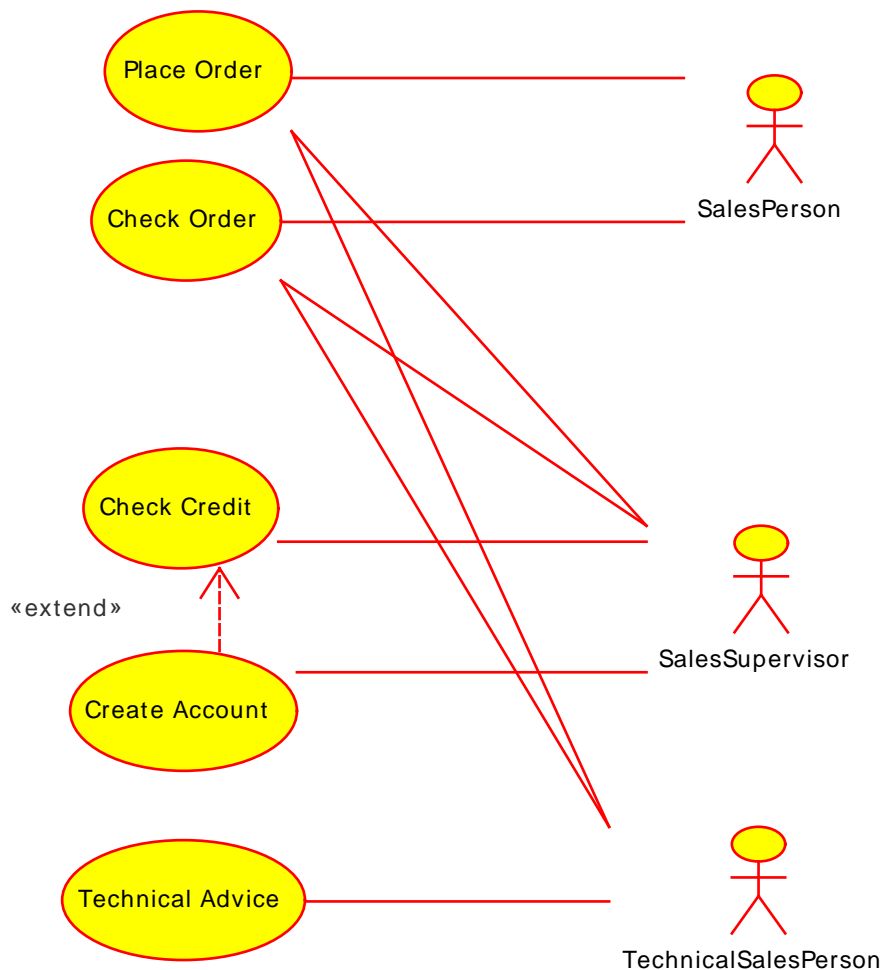
While this is logically correct in that it accurately captures the information given in the text, the number of associations in the diagram makes it difficult to read. It provides no more insight into the system than does the textual description.

In the UML, the symbol for a generalisation is an arrow:

Figure 3.7. Use case generalisation
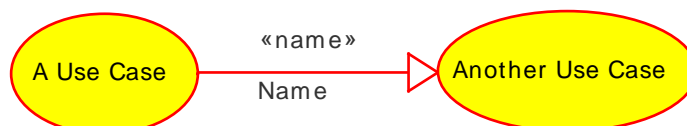
The arrowhead points to the more general entity. If we take account of the generalisation relationships present in the "Sales" example, we reach a model like this:

Figure 3.8. Use case example, with generalisation
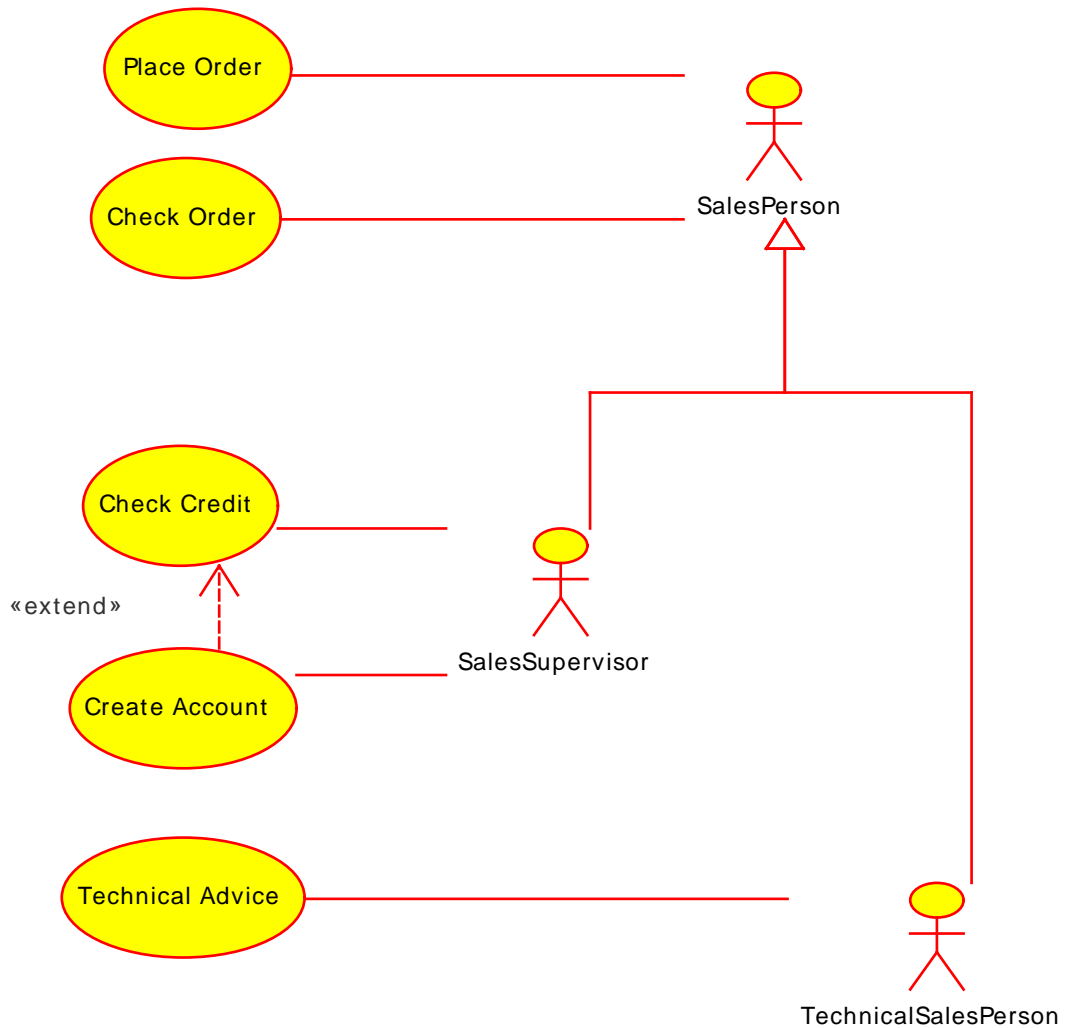
This model does not have to show that "Technical Salesperson" and "Sales Supervisor" can check orders and place orders, because this is implied by their being sub-classes (specialisations) of "Salesperson". Not only is this model easier to read, it gives a more immediate insight into the system being analysed.

You will not always be able to simplify a use case model using generalisation, but you should be on the alert for the opportunity to.

In the example above, "Salesperson" was an actor in its own right, as well as being a generalisation of other actors. That is, we would have identified "Salesperson" as an actor with or without generalisation. Sometimes, however, it is appropriate to "invent" actors simply to stand as generalisations of other actors, with the purpose of simplifying the model. These actors are referred to as "abstract", because they abstract, or simplify, a system. An abstract actor is a special case of an abstract class. We shall have more to say about abstract classes in the next unit.

A simple example

This is an example of a complete, simple use case diagram. It is based on the "Sales" example presented earlier. We will start with a description of the business, then present the use case diagram and the textual specification of the individual use cases. Note that a use case model is incomplete without this specification, either in plain text or something else.

A use case example

A retail business wishes to automate some of its sales procedures. The retailer buys items in bulk from various manufacturers and re-sells them to the public at a profit. Preliminary interviews reveal that there are number of staff roles in the Sales department. A salesperson can place orders on behalf of customers and check the status of these orders. A technical salesperson has the same duties, but additionally is able to provide customers with detailed technical advice (which we would not expect an ordinary salesperson to be able to do). A sales supervisor is a salesperson, with the additional responsibility of creating new customer accounts and checking their credit-worthiness. A dispatcher is responsible for collecting the goods ordered from the warehouse and packing them for dispatch to the customer. To assist in this operation, the computer system should be able to produce a list of unpacked orders as well as delete the orders from the list that the dispatcher has packed. All staff are able to find general details of the products stocked, including stock levels and locations in the warehouse. A re-ordering clerk is responsible for finding out which products are out of stock in the warehouse, and placing orders for these products from the manufacturers. If these products are required to satisfy an outstanding order, they are considered to be "priority" products, and are ordered first. The system should be able to advise the re-order clerk of which products are "priority" products. A stock clerk is responsible for placing items that arrive from manufacturers in their correct places in the warehouse. To do this the clerk needs to be able to find the correct warehouse location for each product from the computer system. Currently, the same person in the business plays the roles of stock clerks and re-order clerk.

Figure 3.9, "A full example", shows the associated use case diagram.

Figure 3.9. A full example
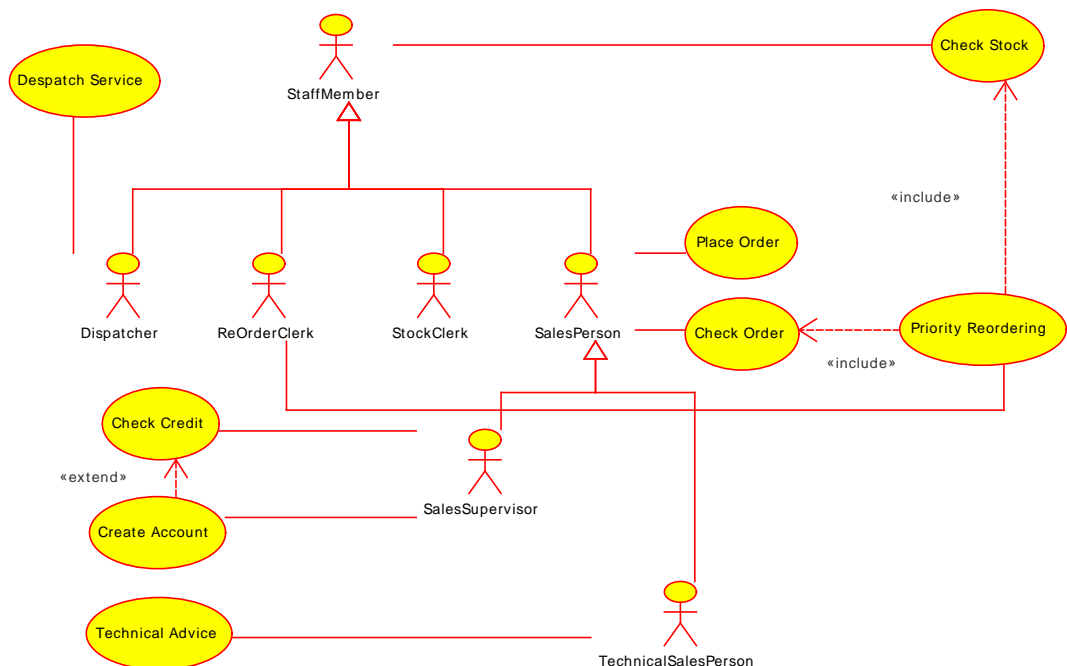
Brief use case specifications

- **Check Stock**: Allows a user to check the levels of stock of any item held in the warehouse, and where that item is shelved. A particularly important scenario is that of obtaining a list a stock items for which the stock level is zero, that is, of which there is no stock in the warehouse.

- **Place Order**: A salesperson places an order on behalf of a client. This has the effect of making information about the order available to Dispatch Service. The order remains on the system until it has been packed and dispatched.

- **Dispatch Service**: Allows a list of outstanding orders to be obtained, and updated when orders are packed. An order is not available to this service if it cannot be satisfied because there is not enough stock in the warehouse.

- **Priority Reordering**: Obtains a list of items that need to be re-ordered urgently, as the business cannot satisfy its own orders without them. This use case makes use of Check Stock (to determine if an item is out of stock) and Check Order (to determine what stock is required to satisfy all current orders)

- **Check Order**: Obtains details about any outstanding orders, including what stock items are required to satisfy the order. This use case is used by salespeople to advise customers, and by the Priority Reorder use case to determine which items of stock must be replaced quickly.

- **Check Credit**: Used to find out whether it is safe to extend credit facilities to a client. This use case refers to external credit reference agencies (not shown).

- **Create Account**: Used to register a new customer. If a customer asks for credit facilities, this use case includes Check Credit. Otherwise it doesn't have to.

- **Technical Advice**: Provides technical specifications for selected products. Used by technical sales staff to provide advice to customers.

Some hints and warnings

Here are a few general hints, and warnings, concerning use case modeling.

- It will usually be necessary to modify and refine a use case model; even an experienced analyst will accept that the first attempt at such a model is unlikely to be optimal.

- A blank sheet of paper (or a blank computer screen) is not a good thing to be looking at when trying to identify use cases or actors. It is better to start by putting down a large number of potential use cases and actors, and perhaps remove or merge some of them later. The early stages of use case analysis can usefully be treated as a "brainstorming" procedure in which large numbers of ideas are floated, only some of which later turn out to be useful.

- A use case should usually provide a discrete, testable service to at least one actor. This makes it possible to implement and test use cases independently. The UML (version 1.3) specification says, "A pragmatic rule of use when defining use cases is that each use case should yield some kind of observable result of value to (at least) one of its actors".

- Use case modeling is subject to the phenomenon known as "analysis paralysis". This is a tendency to concentrate on one small part of a model, adding increasing amounts of detail, while neglecting the broader view. If you develop part of a model to a high level of detail, perhaps over an extended period of time, you will have an emotional disincentive to delete it later should it prove beneficial to do so. This tends to bring the whole process to a halt, as the analyst struggles vainly to complete a model that is too incorrect to be amenable to completion. The correct procedure is to start with a broad outline, and add detail later. For example, if you intend to document a use case with an activity diagram or some other model, it is best to avoid doing this until most of the use cases and actors are defined.

- When describing your use cases, however you choose to do it, you may well find that the process of description leads you to challenge your choice of use cases or actors. You should take this challenge seriously, and modify the model if necessary.

- You should consider using generalisation relationships to simplify a model, buts it is usually best to first identify most of the actors and use cases.

Review

Questions

Review Question 1

What is *requirements engineering*?

A discussion of this question can be found at the end of this chapter.

Review Question 2

Supply the seven steps that make up requirements engineering and briefly describe them.

A discussion of this question can be found at the end of this chapter.

Review Question 3

What advice do you have to offer between the two different representations available for *actors* in a use case?

A discussion of this question can be found at the end of this chapter.

Review Question 4

Construct a use case model that shows the requirements of a computer system that will automate the services of a large lending library. Assume that the library has separate adult and child services, orders its own books stocks, can obtain books from other libraries on request, has a catalogue on public access, and charges fines for overdue returns. Try to envisage all the services and users of the library, and capture them all in the diagram. Do not include services that don't relate to books (e.g., Internet access).

A discussion of this question can be found at the end of this chapter.

Review Question 5

Can this use case model:

Figure 3.10. Without generalisation
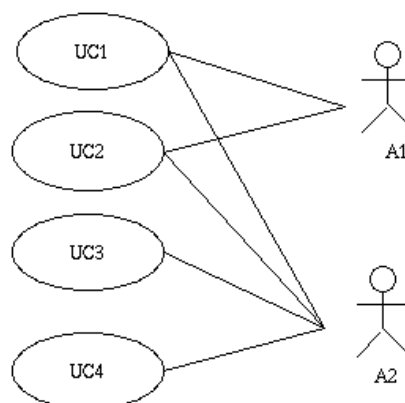
be simplified using generalisation into the model shown below?

Figure 3.11. With generalisation
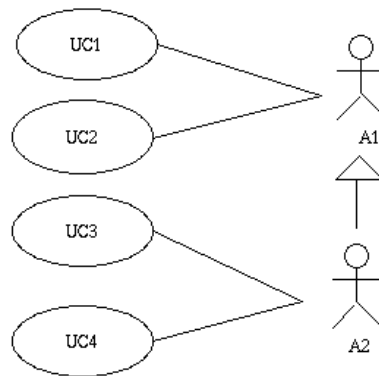
What reasons are there for thinking one way or the other?

A discussion of this question can be found at the end of this chapter.

Review Question 6

In a real-world design exercise, it can often be difficult to obtain the information needed to construct a complete use-case model. Why? (Try to think of at least ten reasons, and possible ways the problems can be overcome). It may help to refer to some general software-engineering books, like Sommerville for information.

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Requirements engineering is concerned with discovering *what* it is that should be developed. Its goal is to develop the software's *specification*.

Discussion of Review Question 2

1. **Inception**, where the developers attempt to gain an understanding of the software and the problems that it has to solve.

2. **Elicitation** attempts to propose solutions to the problems that the software has to solve.

3. **Elaboration** expands the information from the previous two stages into an *analysis model*.

4. **Negotiation** is used to resolve any conflicts between the stakeholders in the project.

5. **Specification** involves producing the final specification from the previous steps.

6. **Validation** concerns itself with ensuring that the specification is of a high enough quality to be useful.

7. **Management** controls and tracks changes made to the specification.

Discussion of Review Question 3

The users of a system can be divided into two general classes: human users and other computer systems. Both of these are represented in use case diagrams. Representing human users using the standard stick figure can be very helpful, while representing other external computer systems as objects

with the «actor» stereotype can help to make a useful distinction between people and automated systems.

Discussion of Review Question 4

There is no single correct answer to this question. You are encouraged to discuss your solution with your tutor and/or your classmates.

Discussion of Review Question 5

If the actor A2 is genuinely a sub-type of A1, then the generalisation shown is logically correct. However, it is not possible to infer whether a generalisation exists from what associations an entity exhibits. Generalisation exists when one entity inherits all the properties of another, not just its associations.

Discussion of Review Question 6

There are many reasons why use-cases can be difficult to construct, and many of these reasons are related to requirements never being completely stable. Some examples to think about:

Use cases are written from the software's point of view, and not that of the actors. For instance, use cases should always describe a user's goal, and *not* a function of the software.

How the software is interacted with is poorly understood. This easily happens when the requirements for the software are novel, or are themselves poorly understood.

Stakeholders disagree. If the clients wanting the software cannot agree on what they want from the software, use-cases cannot be constructed.

Chapter 4. An Introduction to Analysis and Design

Objectives

At the end of this chapter you should be able to:

• Describe the difference between analysis and design.

• Describe how analysis and design are related.

• Describe what a software model is.

• Provide examples of models used in software analysis and design.

Introduction

The previous chapters have hopefully shown you that unless software systems are well thought out and developed using a systematic approach, they can easily become catastrophic failures. Software process models have evolved to solve this, guiding the management of software projects. This chapter discusses *system analysis* and *system design*, which are used during the generic framework activities of *communication* and *modelling* (although they may be found in the others). Analysis and design helps us to understand the problem domain the software must solve, as well as the software itself.

System analysis

The previous chapter — Chapter 3, *Requirements Engineering* — discussed requirements engineering, which is concerned with *what* the software is required to do. Analysis is a tool to help with this: the analyst must determine — from the problem descriptions and incomplete and informal requirements — what it is that the software should do. This is can be an exciting and often challenging task, and focuses on describing *what* the problem is, rather than on *how* it will be solved.

In many cases, analysts build models of the existing system to help software engineers understand how the customers requesting the software are currently dealing with the problem the software should solve.

The result of the analysis is a system specification, a detailed, logical description of either the existing system, or of the new system. For the new system, this specification must satisfy the software requirements. This description must aim to fill in any gaps or ambiguities and make explicit any assumptions in the requirements.

System design

The software / system designer uses the system's specification as a starting point to determine *how* the system should achieve its requirements. Once a particular solution has been adopted, the specification is expanded and modified to clarify what must still be defined in order to be able to achieve the requirements.

The product of this activity is a complete, detailed software design.

The relation between analysis and design

Analysis and design are related activities. As such they may be thought of as two parts of the single process of converting requirements into a clear, complete and coherent software system. Thus, many

of the special techniques (and computer-based tools) used during analysis carry through into design. It is for this reason that the two activities are studied together.

The following chapters of this module introduce several techniques needed for software analysis and design. These are embodied in the idea of *models*.

Introduction to models

The major artifacts of software analysis and design are *models*, a **model** being a representation of one or more aspects of the system.

The most obvious example of such a model is a map. It represents a geographical location and includes important roads, buildings and railway lines. A map does not include all detail for that location — the amount of given detail depends on the scale of a map. A map of *Great Britain*, for example, is unlikely to show every street in the country. A town map, on the other hand, deliberately shows all streets and street names, but does not show the location of all trees, plants and animals in that town.

Other examples include an architect's drawings for a building; a wiring diagram for a micro-chip; the sheet music representing a score of classical music composition. Models represent important features of the thing they model, and can be used to understand and re-create the object being modelled, the building or chip or music itself.

Models *simplify* what they model to a level of detail appropriate for its readers to understand. A road map represents enough information for a person to navigate (by car) between points on the map. The trees and plants are left out of the map because that sort of detail is inappropriate and unnecessary when navigating along streets.

This raises an important point. The model used should be appropriate to the problem. Or, rather, different models should be used for different things. A road map is good when travelling by car, but is less useful when walking in the country-side, where information about footpaths, field boundaries and contours is needed. Similarly, a micro-chip wiring diagram does not have the detail necessary to build a whole computer with disk-drives, monitors and stereo sound. A different model is needed to build a computer.

In addition to this, a person may need training before they can understand the model. Maps are reasonably clear, but even then not everyone can use them. Much specialist training is needed to understand an architect's designs; similarly, training is needed to understand the models of different aspects of a software system.

These examples serve to point the way in which we approach models for analysis and design. The main points are:

• Models are a simplified representation of a system

• Different models are used to represent different aspects of a system

• You need to learn how to understand, construct and use each particular kind of model

The method used to produce a model is called a **modelling technique**. Software which assists us to employ a modelling technique is called a **modelling tool**.

Definition of the term "model"

Two definitions of "model" from various software engineers are:

> A model is a qualitative or quantitative representation of a process or endeavour that shows the effects of those factors which are significant for the purposes being considered.

— H. Chestnut, "Systems Engineering Tools", Publisher: John Wiley, New York,
1965

A model is the explicit interpretation of ones understanding of a situation, or merely
of ones ideas about that situation. It can be expressed in mathematics, symbols or
words, but it is essentially a description of entities and the relationships between
them. It may be prescriptive or illustrative, but above all it must be useful.
— Brian Wilson, "System: Concepts, Methodologies and Applications", p.8,
Publisher: John Wiley, New York, 1984

The properties of models

Before we can decide on what sort of models to use as an aid to software engineering, we should first
determine what it is that we want the model to do for us.

The requirements for any given software system can be immensely complicated. In order to manage
this complexity the system needs to be subdivided into parts, and we need to focus on the requirements
for each of these parts. Having modelled a part of the system, the customer needs to confirm that the
model captures what it is that they want developed. Thus the aim of this whole process is to produce
a complete model of the system understandable by designers and programmers, and, where possible,
the customer as well.

Many iterative and incremental process models begin development on only a portion of the software's
final requirements, and perform analysis and design on this limited functionality. This software
product will then be shown to the customer, who will then request changes (if any) before development
on the next software increment begins. This next iteration of work will include further requirements
engineering, analysis and design.

So a good modelling technique should help in the following ways:

• it considers only part of the system, focusing on particular aspects of the system

• it helps to organise the analyst's ideas

• the customer is able to understand the model, and is able to provide feedback on it

• omissions and errors should be easy to identify

• designers and programmers can produce the system from the models

Since a single model only represents part of the system, it is clear that we will need several models to
help analyse and design a complete system. Given this, there are certain features which models must
have in common in order to meet the above objectives.

Model properties: maintainability and disposability

In the course of developing a system, the analyst's understanding of the system will change in light of
the analysis they do. The same is probably true of the customer. So in the course of analysing a system,
the analyst will often make several versions of each type of model — either building upon earlier ones
or even throwing old ones away and starting again. It makes sense then to have models which are
reasonably cheap and can be developed and maintained without too much cost and time overhead.

For this reason the models often take the form of diagrams, pictures and text which can be printed
on paper or drawn on a white-board. This makes them very cheap! Occasionally, more sophisticated
models are used, such as software prototypes in the prototyping life-cycle model.

The models used should also be easy to change. Re-writing or re-drawing models by hand is tedious,
and CASE software tools (Computer Aided Software Engineering tools, see Chapter 2, *Process and*

Model) can be used to relieve the burden of producing models. More fundamentally though, the models themselves should be such that adding or removing detail does not have many repercussions on other, distant parts of the model. In most cases this means that the model should not represent the same piece of information in two different places. This property of reducing multiple representations of the same thing is called **minimal redundancy**. The property of changes to part of a model having minimal repercussions on other parts is called **low viscosity** or **robustness to changes**. A good model will usually have both minimal redundancy and be robust to changes.

Model properties: graphics and text

Having decided to produce models on paper, we have the choice between graphical representations of the system or textual descriptions.

It is possible to describe a system entirely in words. There are some problems with this, the biggest being that for a large system, a huge number of words will be needed. A developer trying to find technical descriptions and design decisions would be lost in a maze of words. It would also require a huge time investment for the customer to check over such a document. This is clearly not practical.

Instead, for many modelling techniques, graphics make up the largest part of the model. A diagram, by using good notation, can represent a large amount of information in a form that is easy to grasp and, more usefully, can be put on a single sheet of paper.

So what characterises good notation? Some suggestions would be that each object in the diagram means something so that there is not much unnecessary clutter caused by objects with little or no meaning. Also, the objects themselves should have well-defined features so that there is little or no ambiguity. Different types of features of a system should be represented by different symbols, so that it is immediately clear what the parts of the diagram mean.

Using graphics does not mean that no text is used at all. In some places text is the best thing to use; indeed, graphics and text can play a supporting role for each other, clarifying or explaining portions which the other would be too tedious to use (such as with use case diagrams, described in Chapter 3, *Requirements Engineering*).

Model properties: comprehension

As discussed at the beginning of this section, the models need to be understood by many people: the customers, the developers, as well as the analysts and designers themselves. If it takes a long time to explain what the models mean, then much unnecessary time will be waisted. Good models should be clear and simple.

The ideal is for models to be well drawn and easy to understand — no explanation of them should be necessary at all. As a rule, a model drawn by an analyst which the analyst themself finds both complicated and unclear *will* be difficult to understand by others.

The main models of traditional analysis and design

Having discussed both what models are and what we expect of them, we will now introduce the models that you will be studying in the following chapters. This section is meant as an overview, so do not worry if you do not understand something. You will be introduced to each type of model in far more detail in one of the next units.

There are three key aspects which need to be considered when attempting to understand a software system:

• the data needed in the system,

- the processes or functions which the system performs

- what events occur and what changes are made over time

It is useful, therefore, to have a different modelling techniques to focus on each of these areas. Three corresponding modelling techniques are:

- Class models

- Data-flow diagrams

- Sequence diagrams

Each of these modelling techniques is a diagram-based technique with accompanying text (either on the diagram, or to supplement the diagram).

The example models presented in the following three sections have been constructed for a particular software system example. The example system is that of an estate agent that organises the buying and selling of houses. Each of the diagrams will represent a different aspect of this system.

Class model

A **class model** is used to represent the data structure of a system. In other words, the focus of this model is the data needed in the system. The data is represented as classes and relationships between classes.

An example class model is given in Figure 4.1, "An example class model of an estate agency".

Figure 4.1. An example class model of an estate agency
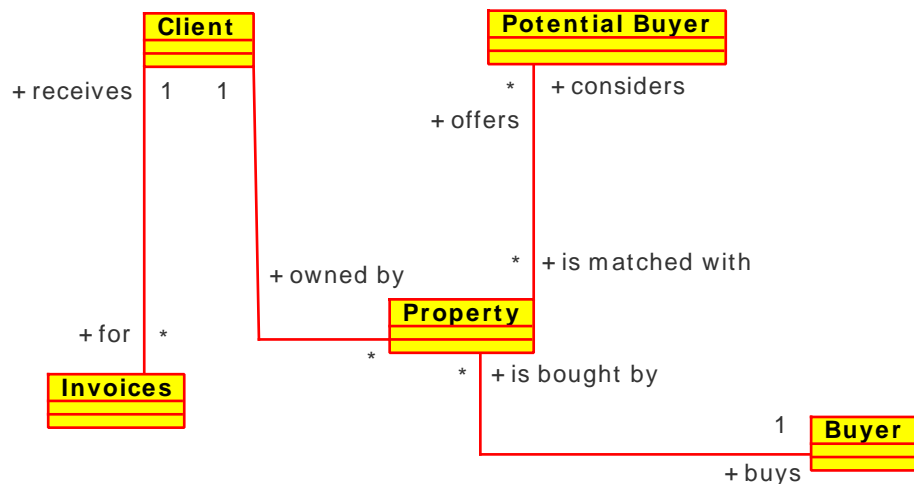
Although no details about class models have been given, it should already be possible for you to understand parts of the diagram — for example, that boxes correspond to classes such as "Client" and "Property", and the lines represent relationships between them. Class models are a very important modelling technique, since the data and relationships of a system are least likely to change significantly. These diagrams are often constructed early on, and the other models built up from them.

Data-flow diagram

Data-flow diagrams (DFD) represent the processes or functions which the system performs, and how data flows between processes and in and out of the system. See Figure 4.2, "An example of a data-flow diagram".

Figure 4.2. An example of a data-flow diagram

D1 is maintained as part of the same system by the keepers.
It could be replaced by the keepers as an external entity.
Keepers could then receive goods delivered from 3.

D3 needs to be maintained in some sensible fashion

Data-flow diagrams represent a somewhat more complicated modelling technique than class models. The named boxes are processes representing actions; the named arrows represent data moving between the processes. Some boxes represent data stores where data flows either to or from. The bubbles are external entities that provide data to the system and receive the output data.

This single diagram does not demonstrate an important feature of data-flow diagrams — their hierarchical organisation. Complex processes (such as "2 Order Supplies") can be iteratively broken into separate processes until the entire system has been described in full detail. Do not worry about the complexities of data-flow diagrams, you will learn the details of this technique in Chapter 6, *Data-Flow Diagrams*.

Sequence diagrams

A **sequence diagram** shows how instances of various classes (taken from, say, a class model) interact with each other over time.

The diagram should be read from top to bottom, which is the direction in which time progresses. The boxes represent not classes, but objects (which is notated using the colons in front of the class names).

Figure 4.3. An example of a sequence diagram

Useful points

All three models represent the same system, albeit different aspects of that system. This is in the same way that a drawing of a human skeleton (a "bone" model) and a model of human musculature differ from one another, but still both represent the human body.

Class diagrams, data-flow diagrams, and sequence diagrams, must relate to each other and be consistent with one another. Though they represent different things, there is a correlation between the various parts of the different diagrams. If these parts do not match up when the diagrams have been completed then it is a sign that either something has been omitted from (at least) one of the diagrams, or that they are not correct. You will learn about the relationships between the different kinds of diagrams in later chapters.

With regards to producing these diagrams for a particular software system, remember that there is no "correct" version of these models. Two different systems analysts may look at the same system and produce two different models. This is almost to be expected, since people approach the analysis of a software system from their own previous experience and, in some sense, their individual perspective gets built into the system.

Just because there is no "correct" model does not mean that all models are equally valid or equally useful. In most cases it takes many attempts to produce a "good" model. The first things written down when developing a model merely act as somewhere to start. As the model is built up the analyst's knowledge of the system increases and they may realise that some details have been omitted or that the original structure is unsuitable or unfeasible. Analysts must be prepared to insert new details, re-draw existing parts of the diagram and even, on occasion, throw away the current model and begin anew.

On modelling notation and software

Class and sequence diagrams will be drawn using the UML. The above diagrams were drawn using the Umbrello software package [http://uml.sourceforge.net], although other packages exist. You may want to explore:

- Dia [http://dia-installer.sourceforge.net]

- ArgoUML [http://argouml.tigris.org]

The SSADM notation is used for data-flow diagrams (since this is not a part of the UML). The freely available Dia package provides support for data-flow diagrams.

The benefits of using formal models

Using models in the software engineering process provides the following benefits:

- Models provides a pictorial, representation of the system, thereby providing the basis for good communication between software engineers and our customers. They are easy to draw and update, as well as being easy to check.

- Any individual model manages complexity through abstraction, by concentrating on one aspect of the system, leaving other aspects of the system to be modeled separately.

- Models impose structure on the information, providing a clear and concise representation that makes the information and its interrelationships easier to understand.

- The techniques by which models are constructed assist in highlighting areas where analysis may be incomplete.

Case studies

A case study is a short text description of an imaginary software system. It is often written in an informal manner and may contain ambiguities. As such, it resembles information that may be gathered

during requirements engineering. The following "Estate Agency case study" is the case study used to produce the examples for the above diagrams.

Estate Agency case study

Clients wishing to put their property on the market visit the estate agent, who will take details of their house, flat or bungalow and enter them on a card which is filed according to the area, price range and type of property.

Potential buyers complete a similar type of card which is filed by buyer name in an A4 binder.

Weekly, the estate agent matches the potential buyer's requirements with the available properties and sends them the details of selected properties.

When a sale is completed, the buyer confirms that the contracts have been exchanged, client details are removed from the property file, and an invoice is sent to the client. The client receives the top copy of a three part set, with the other two copies being filed.

On receipt of the payment the invoice copies are stamped and archived. Invoices are checked on a monthly basis and for those accounts not settled within two months a reminder (the third copy of the invoice) is sent to the client.

Review

Questions

Review Question 1

Contrast *analysis* and *design*. How are they similar, how do they differ?

A discussion of this question can be found at the end of this chapter.

Review Question 2

What role do models play in analysis and design?

A discussion of this question can be found at the end of this chapter.

Review Question 3

Briefly outline the various properties of a model.

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Analysis and design are related activities, concerned with (respectively) *what* the problem is, and *how* it will be resolved. Because of this relationship, many of the modelling techniques used in analysis are themselves used in design, and vice versa.

One large difference between them is that analysis does *not* consider any implementation details: it is merely concerned with description of existing problems. Analysis produces a specification of *the existing problem* or the *existing solution* to the problem.

Design itself can begin as a description of a *solution* to the problem, but will progress into implementation details. It also produces a specification, but this specifies the *solution* to the problem.

Discussion of Review Question 2

Models provide one of the primary means of communication of information during software engineering, especially during the activities of analysis and design. They allow us to *simplify* the information we wish to convey, and thus can *highlight* specific portions of the software system, such as the static data design (class model), user interactions (use-case diagrams) and the flow of data (data-flow diagrams). They are a tool for abstraction (see Chapter 7, *Design*).

Discussion of Review Question 3

Models have the following properties:

- they consider only part of the system, focusing on particular aspects of the system

- they help to organise the analyst's ideas

- if the customer is able to understand the model, and is able to provide feedback on it

- omissions and errors should be easy to identify

- designers and programmers can produce the system from the models

Chapter 5. Object-oriented Analysis and Design

Objectives

At the end of this chapter you should be able to:

• Create class models of systems that can be encapsulated in one diagram.

• Simplify a model using generalisation and abstract classes.

• Describe how a more complex model can be specified, and how modelling software maintains design integrity in such a complex system.

• Read and describe class models created by other designers.

• Be able to relate class diagrams to equivalent program outlines.

• Specify an object model using CRC cards.

• Describe how objects change over time.

Introduction

This chapter discusses object-oriented modelling methods, that is, the representation of a software system in terms of classes and their inter-relationships. Class modelling is the most fundamental aspect of object-oriented analysis and design, and its mastery is crucial for anyone who intends to use object-oriented techniques. Class modelling is useful to both the analysis and design disciplines: in analysis, classes and the relationships between them will specify the problem. In design, it specifies the software.

Class modelling is philosophically complex, but practically straightforward. However, it rests on a few important philosophical concepts, and it is vital that these concepts are mastered before doing any practical work. When students fail to make progress with real design exercises, it is commonly because they have not developed a proper understanding of the basic principles. These principles cannot be learned by rote, they must be understood through exercise and application.

Class modelling is used throughout object-oriented development, from analysis through to implementation and testing. The class model is the "skeleton" on which the "flesh" of program code is constructed. During the development of a system the class model is modified and refined, and often several different models will be used for the same software system, each model seeing the system form a different standpoint, as will be discussed further below.

Modelling standpoints

Class modelling is a flexible technique with a number of applications. In this it should be contrasted with use case modelling; although the purpose of use case modelling is not specified in the UML, in practice it is usually employed as part of a requirements elicitation exercise (as discussed in previous chapters). Class modelling, on the other hand, is used throughout an object-oriented analysis and design process, from requirements engineering through to programming.

The designer should have in mind at each stage of development what the class model is being used for. Neglect of this principle leads to poor design work. If you believe you have produced a class model

that represents "the system" (whatever the system happens to be), you are quite likely mistaken. A single class model at best represents a particular *standpoint* on the system.

Design standpoints represent the different uses we make of modelling. We can identify three obvious standpoints.

- A *logical design standpoint*: the model shows how a system should function logically, or how its components interact in a logical sense.

- A *specification standpoint*: the model shows how the system is broken into manageable components, and how those components interact.

- An *implementation standpoint*: the model shows how the system is structured, or how it will be structured in the future.

Although these standpoints are not completely distinct, they do embody differences in how the software system should be modelled. Specifically, you should see that analysis models will be more of a logical standpoint, while design models will, ultimately, be an implementation standpoint.

As design progresses the standpoint of the designers will change. Initially — especially if the system to be developed is replacing some other system or manual process — the designers will be concerned mostly with how the system is currently structured. This existing structure may well not be the most efficient or logical, but it is necessary to understand the operation and context of the existing system before designing a replacement. Later, in the design process, the designers will be concerned with the logical structure of the system. That is, regardless of how an existing system may be implemented, the designers consider what the simplest, most effective, logical organisation of the new system is. Still later, the designers will need to refine the model into a design that can be implemented. Here they will be concerned with specification and implementation standpoints.

The UML provides a set of modelling symbols and terminology and defines the concepts that they embody. The first job of the novice designer is to learn this notation and come to understand the concepts. The UML does not specify how these concepts are to be employed. In other words, it does not provide a different set of symbols for logical and implementation standpoints; it is up to the designer to ensure that his or her intentions are clear.

Classes and objects

This section discusses the nature of classes and objects. We begin with the philosophical background of classification and move onto descriptions of objects and classes that will be useful for software and system design.

Classification

The idea of *classification* is not a new one; philosophers have been studying the concept for at least two thousand years. We classify things all the time: we classify people by gender or by age; we classify motor vehicles by size, passenger capacity, and so on; we classify businesses into publicly and privately owned entities, amongst others. When we classify something we are saying that it has properties in common with others of its type. The idea is, at heart, a simple one. We assume that if we know something about the class, then that fact is true of all the members of the class as well. In writing a computer program that manipulates, say, customers' bank accounts, we need to be concerned first of all with the things that all bank accounts will have in common. We do not want to be concerned with individual customers unless there is something about them that is different to the class as a whole. This is a profoundly important and simplifying principle.

By being members of a class it is assumed that an object shares the general properties of the class. For example, all members of the class "human being" have certain properties in common; although there are a great many different human beings in the world, they are sufficiently similar that there should be no difficulty distinguishing between any human being and, for example, a cheese sandwich.

Classification in object-oriented design

In object-oriented design, we classify all the objects that a system should know about. That is, we specifying classes that collectively describe all the behaviour of the system. Although we speak of object-oriented design, we are mostly interested in *classes*, rather than individual *objects*. **Class modelling** is the process of assigning classes, and describing their inter-relationships.

Class modelling works in software design because it allows complex systems to be described in such a way that the complexity becomes manageable. And all interesting (and most useful) software systems will be complex.

The "things" that a software system will process, recognise and serve are, by definition, objects. Users of the system, keyboards, screens, bank accounts, stock items, printouts, and so on, are all objects. By classifying these objects, that is, by creating and managing classes, the designers of the system impose some control on the complexity of the system. Although a complex system may manage many millions of objects, it will frequently manage orders of magnitude fewer classes, perhaps fewer than one hundred.

A definition of *class* and *object*

For the purposes of object-oriented modelling, an object can be defined as follows:

> An **object** is a self-contained entity with well-defined, recognisable properties (*attributes*) and behaviour (*operations*). It can interact with other objects.

This leads to the following definition of a class:

> A **class** embodies the properties and behaviour that a collection of objects have in common.

Examples of classes

Having presented this philosophical discussion, you may be expecting classes themselves to be philosophical. In fact, they are usually not. Suppose we are designing a system to manage the work of a lending library. Some of the classes that will need to be considered are:

• Book

• Member

• Reader

• Borrower

• Loan

• Fine

• Barcode

• Return date

Of course, on more detailed inspection these may turn out to be inappropriate, and no doubt many other classes will have to be considered, but these initial ideas are appropriate classes because:

• they are self-contained with well-defined behaviour and properties

• each will have some instances (objects) that have a great deal in common

- it is obvious that each has an important role to play in the system

A further key point is that some classes may in fact have sub-classes. For example, "Reader" and "Borrower" may in fact be types of "Member".

Some thoughts on the relationship between classes and objects

Here are some concepts to consider when thinking about classes and objects:

- In English we use the words "is a" to mean several different things. For example, in "Rover is a dog" we are saying that Rover is an instance of class dog. When we say "A dog is a mammal" we are saying that "dog" is a sub-classes of "mammal", that is, all members of class dog share properties with members of class mammal. Therefore, in describing the relationship between classes and objects, you should take some trouble to use unambiguous terms like "is an instance of" and "is a subclass of".

- A class may have any number of instances, including one and zero. Classes can be defined that have zero members in practice and in principle. Student are often particularly reluctant to use classes that have only one instance. The "Internet", for example, is the one and only instance of class "Internet". Again, this may seem odd, but *class modelling* is about *classes*, not *individuals*.

Concrete and conceptual classes

Previously we looked at some examples of classes that might be appropriate for a library computer system. You should be able to recognise an important sub-grouping of these classes. Classes such as "Book" and "Member" are physical or concrete classes; they correspond to "real" things in the physical world. However, the classes "Loan" and "Return date" are different "types" of class. They do not correspond to real, physical entities at all. We will refer to classes of this sort as "conceptual" classes. Many student mistakenly use the term "abstract" here; this should be avoided because "abstract class" has a quite different meaning, which will be explained later.

Some components of a software system have both concrete and conceptual representation. Consider the "Book" class: The conceptual book "War and Peace" by Tolstoy exists independently of the paper and binding of a real physical book; indeed the conceptual book may be said to exist if the only text in existence was in a computer's memory. In other words, in a sense the book exists outside of its physical representation. At the same time, a lending library loans physical, not conceptual, books. They are real entities made of paper, with barcodes for scanning, and so on.

This is not purely an academic distinction. Large, complex systems (particularly object-oriented databases) may be impossible to implement unless this subtle distinction is understood. For example, if the library's system only represents physical books, it has to store all information about each book in each instance of the "book" class. But if the library has, say, ten copies of each book, then most of this information is duplicated in each object, which is a waste of memory and can cause data inconsistencies. This problem always arises when there is a distinction between physical and conceptual representations of the same thing, and when there are multiple, similar physical instances of the conceptual item.

This is not an issue on which the novice designer should lose sleep; the majority of systems can be designed correctly even if the issue is not understood. However, it will improve the depth of your understanding if you take the trouble to ensure that the sense of this section is understood.

Note

In an implementation, conceptual and physical classes will have different names to one another. For instance, the conceptual class "Book" might be named "BookInformation", while the physical class may retain the name "Book".

Attributes

Attributes are the properties of a class; on the whole they are things that can be measured or observed. For example, an important attribute of the class Animal is "number of legs". Different animals have different numbers of legs, but all animals have the property "number of legs". Even a snake has a number of legs: it just happens to be zero. However, it would make no sense to talk about a plant's having "zero legs". The attribute "number of legs" is not a valid attribute of plants.

Attributes are used rather vaguely in classification in general. For example, biologists distinguish between insects and spiders because insects have the attribute "has six legs" and spiders have the attribute "has eight legs". There is a numerical difference here. However, one of the ways that biologists distinguish between reptiles and mammals is that reptiles are cold-blooded and mammals are warm-blooded. There is no numerical difference in this case.

In class modelling the nature of attributes is formalised. We can say that:

> An **attribute** of an object is a property that has a name, a value and a type. The name and type must be identical for all instances of a class, but the value may be different.

For example, suppose the class "Book" in our library example has attributes "title", "publisher", and "date of publication" (there will, no doubt, be others, but these will do for now). The attribute "title" has a name ("title"), it has a type (probably "text" or "string"). Objects of class "Book" will have values for this attribute, for example, "War and Peace". The attribute "date of publication" has a name, and its type is "Date". Again, objects of class "Book" will be published on different dates, but all objects will have a value for the attribute.

The important point here is that, although all instances of class "Book" have different values for these attributes, they all have the attribute, and they are all the same type. The date of publication of a book will always be a date, and never a height. The title will always be a piece of text and never a colour, and so on.

In some cases the values of attributes distinguishes one object from another. For example, if Rover is a brown dog and Fido is a black dog, we may reasonably say that their "colour" attributes are different. There is no chance of confusing Rover for Fido. However, common sense indicates that if we see two brown dogs, we cannot assume they are the same dog: there are many brown dogs in the world. Indeed, even if all the attributes of two objects are identical, this does not make the objects identical.

There is an interesting philosophical issue behind this; many students have argued (sometime fiercely) in classes that if we really knew all the attributes of a particular class, and two objects had all those attributes in common, then they would of necessity be the same object. For example, two brown dogs may be indistinguishable by colour, but the are distinguishable by position. If two putative objects occupy the same point in space, then they must be identical.

This is all well and good, but irrelevant for this subject. Whatever the philosophical contentions, the principle which object-oriented designers work to is that:

> Two objects are not equal, or identical, just because they have identical attributes.
> Objects are only **identical** to themselves, or things that refer to themselves.

For example, the person Mary Smith is identical to herself, and if Mary Smith is the world-record holder for (say) javelin throwing, then the object "Mary Smith" is identical to the object "world record holder for javelin throwing".

Ignorance of this principle leads to very subtle problems when designs are translated into programs. For example, consider the following portion of a Java program:

```
String response = "HELLO".toLowerCase();
    if (response == "hello")
    {
```

```
System.out.println("They are identical.");
}
else
{
System.out.println("They are NOT identical.");
}
```

This Java snippet creates an object of class String and sets it equal to the lower-case text "hello". The program then tests whether the "response" is equal to the object "hello". Since both objects have the value "hello", we might expect the program to consider them equal. But in fact the program will inform us that the two objects are not equal; the object "hello" and the object "response" have identical attributes, but they are not equal to each other. The correct way to test whether two String objects have equal attributes in Java is to write something like:

```
String response = "HELLO".toLowerCase();
    if (response.equals("hello"))
    {
System.out.println("They are equal.");
}
else
{
System.out.println("They are NOT equal.");
}
```

The **equals** method tests two objects to see if their attributes are equal. The == operator, on the other hand, tests whether the objects themselves are, in fact, the same object.

It is a key principle of object-oriented design that attributes should be simple. The technical term is **atomic**, meaning "not capable of division into smaller components". If an attribute appears to have attributes of its own, then it is not an attribute at all, and is probably an object in its own right. You will find that programmers, of necessity, do not follow this rule. This is because it is standard practice to use classes to extend the functionality of a programming language. In Java, for example, numbers — such as 12 and 4.7 — are atomic, but text strings are represented as objects. In this case it is often necessary to make objects attributes of other objects.

Also, although "has six legs" may be an attribute of insects to a biologist, it will never be an attribute of any class in an object-oriented design. This is because it is not precise enough. A designer would say that the class "insect" has an attribute "number of legs" whose value, by default, is 6. We would write this in a formal way (as will be discussed later):

```
numerOfLegs:integer(6)
```

Having said that, in the initial stages of design, it is inadvisable to be too precise about attributes. In particular, usually the names are established first before the types and values. In some cases the types of attributes will not be determined until writing the program code; however, at this point it must be specified: many programming languages require the types of attributes to be fully specified.

In addition, during design there is no reason why the types of attributes have to match the types that are available to a particular programming language. For example, real numbers (that is numbers with fractional parts) are represented in Java by the types "float" and "double". These are technical terms with no meaning outside of certain programming languages. In design there is nothing to prevent the use of attribute types "number", rather than "float", simply because it is easier for non-specialists to follow. The translation into programming terms can be made later in the development if necessary.

It is important to understand that an object's attributes belong to the object in some way, that is, they tend to be "private". One object cannot automatically obtain, or change, the attributes of another. This

is just another way of saying the objects are self-contained. Part of the process of object-oriented design is determining which of an object's attributes may be read and written by other objects, and which will remain private.

Terminology note

Other terms for "attribute" include "instance variable", "member variable" and "member data", and may differ between programming languages (Common Lisp, for instance, uses the term *slot*). Technically an "instance variable" is not exactly the same as an attribute, but the distinction is not important enough to worry about at this stage. Java programmers tend to use the term "instance variable" rather than "attribute", but "attribute" is preferred by most designers and CASE tools.

Operations

In the last section we considered attributes, which say something about what a class is. *Operations* say what a class does (or can have done to it). When you have fully specified the attributes and operations of a class, then you have specified that class completely (some experts would say that one must specify the class's constraints as well, but that is largely beyond the scope of this course).

For example, the "Clock" class may have the operation "display time". This is something that a clock does. It is not an attribute of class "clock", because it does not tell us anything about objects of class clock that we do not already know.

At the design stage, we tend to assume that all of a class's operations are accessible to other classes, that is, classes can communicate by invoking each others operations. This is called "message passing". As design progresses, we usually find that new operations have to be added that are private to that class, because they are only concerned with internal operation of the class. Remember that classes should be self-contained, and should expose as little as possible of their internal workings.

In the same way that attributes are formalised in object-oriented design, beyond what is true of classification in general, operations are also formalised. An operation has a name, a parameter list, and a return value, much like a function. The parameter list is (essentially) the data elements that will be supplied to the operation. The return value is a piece of data that can be returned to the object that invoked the operation. These issues are not very important in the early stages of design (we may only know the name of the operation), but become important as the design is translated into a program. Note that an operation has access to the object's attributes, and can read or write them, but normally has no access to the attributes of other objects.

Terminology

Other terms for "operation" include "member function", "class function", and "method". Java programmers tend to use the term "method".

Dynamic behaviour and state

A class model, taken alone, is a description of the static structure of a system; this means that it shows the system in a way that is independent of time. In the library system discussed earlier, the individual books may come and go, and individual members for that matter, but the classes "Book" and "Member" persist. To say that a lending libraries has books and members is true, independent of time.

Of course, in reality the system will change over time. This change is reflected in the values of the attributes of objects, and the numbers of instances of each class. The library may begin with 1000 books, and ten years later it may have 10 000 books; but there will never be a time when the class "Book" is not a valid one for the library. Similarly, the addresses and even the names of the members will change. This will be reflected in the values of the attributes of the instances of class "Member".

The sum total of information carried by the attributes of an object is called its **state**. If we know the state of an object, then we know everything there is to know about it. This last statement is true by definition. If in describing an object the designer finds that its attributes are not sufficient to specify its state completely, then they have overlooked some attributes.

If the state of an object is given by the values of all its attributes, then the state of the system if given by the states of all its objects. In other words, the values of all the attributes of all the objects completely specify the state of the system at any given time.

If state is so important, why can it not be represented on a class model? The reason is that there are far better ways to represent changes in state. The UML provides state transition modelling for this purpose. In a full design it may be necessary to provide state transition diagrams for some or all classes, to describe their state changes in detail. The pattern of state change over time is called the **dynamic behaviour** of a system, and modelling this is called **dynamic modelling**. As well as state transition modelling, the UML provides *object interaction diagrams*, *activity diagrams*, *sequence diagrams* and *communication diagrams* for dynamic modelling. One can, of course, describe the dynamic behaviour of classes in plain language as well, and one probably should.

Only objects have state; classes *do not* have a state. This is because a class specifies properties that are true of all objects of that class. However, a class can have a *default* state. This is the set of attribute values that an object of that class will have by default, that is, if not provided with any other information.

UML notation and conventions

Symbol

The diagram below shows the UML symbol for a class formally (left) and as an example (right). The formal detail looks quite daunting, but you will see that most of the information is optional, and in practice a class symbol is quite straightforward.

Figure 5.1. The UML symbol for a class
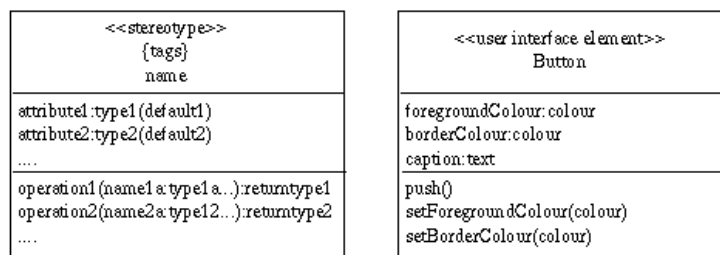
The class symbol has three compartments. The top compartment shows the name of the class, any tags it has, and any stereotype it has. The stereotype of a class is a higher-level category to which it belongs. Stereotypes are used to provide additional structure to a model, and to make it easier to follow. Common stereotypes include «use case» and «actor». Most classes, in practice, will not need stereotypes. The designer is at liberty to create new stereotypes to organise the work, if required. The only tag that we will consider in this module is {abstract}, which will be discussed later.

The middle compartment contains the class's attributes. Each attribute can specify a name, a type and a default value. Only the name is compulsory.

The bottom compartment contains the class's operations. These can be specified in full, with parameter lists and return values, but very often only the name is given. To distinguish operation names from attribute names, it is conventional to write an operation name with pair of brackets after it, like this:

push(). If the operation has a parameter list, then it can be written in the brackets, but, again, this is not compulsory.

The example shows a possible class diagram for a "Button" class. By button we mean a little rectangle that appears on the screen, and responds to mouse clicks, as with a computer GUI rather than a physical button. In this example, the button has the stereotype «user interface elements». It is intended that this stereotype be applied to all classes that represent user interface elements, like list boxes, menus, windows, and so on. Again, this is not compulsory, but it helps structure the model. There are no tags associated with this class, so none are shown.

The "Button" class has a foreground colour and a border colour, and these can be changed. So we have defined two attributes to store the colours. These attributes have type "colour". Most programming languages do not have an inbuilt "colour" data type, and at some point the programmer will have to translate this type name into something that the programming language will understand. This is not an issue for the early stage of design. It is clear what "colour" means, and this is the important factor.

The operations of the class are "push()", meaning simulate what happens when a button is pushed (probably the screen appearance will change), and two operations to change the colours. Why can another class simply not change the attributes to change the colour? It is vital to understand that a class should be, as far as possible, self-contained. It is up to the class to decide whether its colours can be changed or not, and by which other objects. The operations to change that colour will probably change the values of the colour attributes, but it is up to the implementer of the class to decide how this will happen. Another class does not need to know the internal operation of the "Button" class.

The operations that set the colour each have a parameter list with one parameter: the colour to be used.

You will see that the example class is very much simpler than the formal specification. In practice, it is common for classes to start of with very little detail (no parameters, no types), and to be "filled in" as design progresses.

A note on software

The class diagrams were created using a commercial package, *Select SSADM*. Various software packages may support the UML notation to a greater or lesser degree, and so diagrams made using different packages will differ from one another.

Naming conventions

The UML makes certain recommendations about how names of things are to be written, with varying degrees of force. We recommend that you follow the UML naming conventions as if they were all compulsory, because most professional designers do, and you will want your work to be understandable by your colleagues. The naming conventions are also in line with the Java naming conventions, and should be familiar.

• Names of classes are written with an initial capital letter (like this: Button, not like this: button) and have no spaces. To show where a space would be, the next letter is capitalised (like this: BarcodeReader, not like this: Barcode Reader).

• Class names are singular, not plural. It is incorrect to name a class "Books". Although there may be many book objects, there is only one book class.

• Names of attributes and operations start with a lower-case letter, but may have capital letters to indicate where the spaces would be, if the name would normally have spaces.

• Names of operations are followed by brackets () to distinguish them from names of attributes

The prohibition against spaces arises because most programming languages do not allow spaces in their naming convention, and it is modern practice for the final model diagrams to be as close to a program as possible. Some, but not all, CASE packages will enforce these conventions.

Finding classes

The customers of a software engineering project will not think of their work in terms of classes; it is the job of the developers to determine suitable classes with which to structure their work. This job requires experience and intuition, and invariably improves with practice. Here are a few general hints:

- The names of classes are usually nouns. A good place to start is to take a written description of the system and select all the nouns. You will undoubtedly have to remove some classes and add others, but this method provides a place to start.

- It is better to have too many classes to begin with, rather than too few. You can always remove some if you find that they are redundant

- Classes should, as far as possible, correspond to entities that the clients would understand. Computer terms rarely make good class names. You will find that as development progresses you will need to add classes to represent implementation factors, but you should not start off with these. Classes should make sense to non-technical people, at least in the early stages.

- If something has sub-types, or it is a type of something else, it is probably a class

- If something is important, but does not appear to be a class, it might be an attribute of some other class

As you add detail to a design you will discover new classes, and you can add these to the model. You will also often find that you can simplify a design by adding classes, as will be shown.

At all stages of development you should challenge your class model and be prepared to modify it if necessary. It is important to remember that iterative, incremental and evolutionary methods will have you update not only requirements, but the software design (and hence class model) for multiple development iterations. This is especially important for agile process methods.

Relationships between classes

Finding and describing classes is the first stage of class modelling. The next stage is to determine, and show, how classes are related.

In the simplest terms, a relationship between two classes is shown by a line (as it is in an entity relation diagram):

Figure 5.2. A relationship between two classes
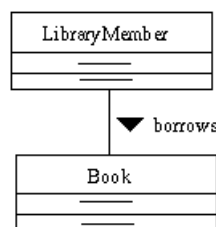
In general such a relationship is referred to as a *link*, *association*, or a *collaboration*.

In the diagram above, the classes are shown with horizontal dashes where the operations and attributes normally go; this is an indication that there are some operations and attributes, but they are not shown on this diagram. It is also permissible to show a class simply as a rectangle, but then the reader does

not know whether the class has attributes and associations and they are not shown, or if it does not have any at all.

The line connecting the two classes has an arrow to indicate the *direction* of the association. Objects of class "LibraryMember" borrow objects of class "Book", not the other way around. This is fairly obvious from the context in this example, and the arrow could have been omitted, but if you are working on a system which is less obvious it does well to include the arrows. Note that links should have names, although you will not always want to show the name on the diagram (to reduce clutter). If you cannot assign a sensible name to a link, it is probably not a real link, or the classes at each end are incorrect. Normally a link should be able to have a relatively simple, short name, which will probably be a verb.

The UML is strict about the positions of arrows. It would be incorrect to draw the arrow in the figure above on the line. Arrows on lines have a different meaning altogether.

The links shown above is an example of a "named association"; it is the most general kind of link. The UML also recognises a number of special-purpose links, which will be described later.

Specifying relationships in detail

If a link is drawn as a simple line, it is assumed to be a *one-to-one association*. In the previous example this would mean that each member of the library can borrow exactly one book, and each book can be borrowed by exactly one member. This is probably not correct. It is good practice to indicate on each end of the link its **multiplicity**, that is, the number of objects associated with the link. It is particularly important to distinguish between "1" and "many" in multiplicities.

We could re-draw the previous example like the figure on the left:

Figure 5.3. Indicating multiplicity
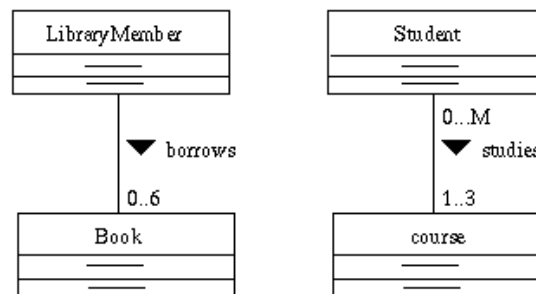
This indicates that at any given time a "LibraryMember" is involved with a "borrows" association with an object of class Book, that is, a library member can borrow up to six books. At the same time, a book can only be on loan to one member.

The figure on the right is an example of a *many-to-many relationship*. A student can be studying one, two, or three courses at a particular time. Each course can be studied by an indeterminate number of students. The symbols "M" stands for "many", and means "some unspecified number". The asterisk (*) can also be used to stand for "many". Note that the UML distinguishes a *many* that may include zero from a many that does not. In the "students" example, the number of students enrolled on a course varies from zero up to many, not from one up to many.

The multiplicity symbols are an example of what the UML specification calls **end ornamentation**. End ornaments are symbols that can be used on the ends of associations to enhance their meanings. Another common end ornament is the **navigability arrow**. This is an arrow drawn on (not alongside) the line and indicates the direction in which the class structure can be traversed for information.

Inheritance

We indicated earlier that object-oriented modelling recognised some specialised types of associations. The first, and most important, of these is the generalisation-specialisation association. You will have encountered this previously in when studying use case modelling; it is now time to examine this relationship in more detail.

As a reminder, the symbol for a generalisation-specialisation relationship is shown below:

Figure 5.4. Generalisation-specialisation represented in the UML
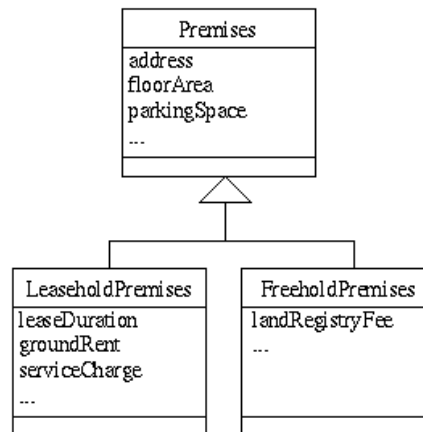
The arrow points towards the most general class. In this example, we are considering part of a system to be used by an estate agency for assisting in finding suitable premises for small businesses to purchase. The class "Premises" represents any premises. Its important attributes are the address, floor size, type of use permitted, amount of parking space, price expected, and so on, but these are not all shown in the diagram for reasons of lack of space (ellipses are used to indicate this). There are two important sub-classes of premises: leasehold premises, and freehold premises. These have all the same properties of "Premises", and some additional ones. For example, leasehold premises will naturally have an address and a size, but they will also have a lease duration, a ground rent, and some others. Similarly, freehold premises will have properties that are not applicable to leasehold premises, or to premises in general.

It would be an error to show the attributes "address", "floor area", and so on, in "LeaseholdPremises" and "FreeholdPremises". These classes will automatically have these properties, as they will inherit them from "Premises". The same mechanism applies to operations: any operations of "Premises" will apply also to its sub-classes and should not be shown again in the diagram except if the sub-classes do different things in these operations.

There may also be sub-classes of "LeaseholdPremises" and "FreeholdPremises". These will inherit the properties of all their super-classes. So classes can form a complex hierarchy. When a sub-class inherits properties from a class other than its own base class, this is called **indirect inheritance**.

Terminology

A variety of different terminology is used to denote different parts of a generalisation-specialisation relationship. Using the above figure as an example, we could say:

- "Premises" is a generalisation of "LeaseholdPremises" and "FreeholdPremises"

- "Premises" is the base class of "LeaseholdPremises" and "FreeholdPremises"

- "Premises" is the super-class of "LeaseholdPremises" and "FreeholdPremises"

- "LeaseholdPremises" and "FreeholdPremises" are sub-classes of "Premises"

- "LeaseholdPremises" and "FreeholdPremises" are specialisations of "Premises"

Multiple inheritance

It is possible for a class to be a sub-class of more than one base class.

For example, "Radio" and "Television" are sub-classes of "ElectronicDevice". "Television" and "Car" are sub-classes of "TaxableProperty". "Television" inherits properties of "ElectronicDevice" and of "TaxableProperty". "Radios" are not taxable, and therefore cannot be a sub-class of "TaxableProperty". "Cars" are not electronic. This is a clear example of multiple inheritance. This example is quite artificial, to illustrate a case where it is difficult to avoid multiple inheritance. In many cases it will be possible to restructure a model to avoid it. Note that some programming languages (including Java) do not allow multiple inheritance, so many designers avoid it as well.

Abstract classes

An **abstract** class is one that can have no *direct* instances.

An abstract class has no direct instances logically, or by definition. But a class is not abstract merely because it happens to have no instances. For example, the class "PersonWhoCanRunAMileInLessThanThreeMinutes" happens to have no instances at present, but there is no logical reason why this should be the case. No doubt careful use of performance-enhancing drugs and twenty years of selective breeding could cause this class to be instantiated. For a class to be abstract it must be logically impossible, not physically impossible, that it be instantiated.

In addition, a class is not abstract merely because it does not correspond to a real-world entity. This mistake is commonly made by novices; it is not unusual to see beginners label classes like "BankAccount" or "Transaction" as abstract because they represent non-physical things. The correct term for these classes, as mentioned earlier, is *conceptual*.

As an example, consider a computer system that manages information about certain products that a business manufactures. The bulk of information about a product (e.g., price, delivery time, warranty duration) is encapsulated in a class called "Product". However, we may choose to create classes to represent the specific products we produce, and have them as sub-classes of "Product". In this case, by definition there are no direct instances of class "Product", but there will be instances of the sub-classes of "Product". Because "Product" is never directly instantiated, it is an abstract class.

What is gained by doing this? In short we have gained a measure of simplicity. By gathering most of the important information about the products into a single class ("Product") we have removed the need to duplicate this information in the different sub-classes of "Product".

Making "Product" abstract has two important benefits. First, the reader of the model will know immediately that "Product" has a particular role in the system, that of providing structure and simplification. Second, the compiler that generates the final program will know that there can be no direct instances of class "Product". This means that (i) it does not have to use memory to store the details of the instances, because there are none, and (ii) it is able to prevent the programmer making some trivial mistakes. Consider this definition in Java:

```
abstract class Product
    {
    //... details go here
    }
```

If the program attempts to create an object of this class later, it can only be because the programmer has made an error. The compiler will flag any such attempt as a compile time error.

An abstract class can have abstract operations. These are operations that are specified, but not implemented. It is up to the subclasses to provide a full implementation of the abstract method.

An abstract class with no-subclasses is more than likely useless. If you find this on your models you have probably made a mistake.

It is possible to indicate that a class is abstract in a class diagram by adding the {abstract} tag beneath the class's name.

Aggregation and composition

All object-oriented designers use generalisation-specialisation relationships all the time. They are vital to good design. The relationships we will consider in this section are defined by the UML, but it is not generally agreed that they are a special kind of association at all.

Aggregation

Aggregation is a general term for any whole-part relationship between objects. Its symbol is a white diamond, like this:

Figure 5.5. The representation of aggregation in the UML
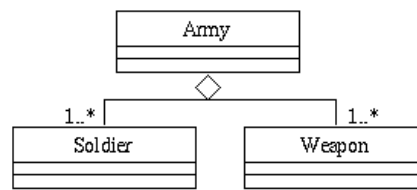
This diagram can be read as "an Army is an aggregate of one or more soldiers and one or more weapons". The aggregation is expressing a loose whole-part relationship between the army and its constituents, where the constituents are not necessarily members of the aggregate. For example, if the army as an entity ceased to exist there would still be soldiers (they would simply be unemployed). This topic has lead to some very heated debate. In practice the distinction between a computer system which works, and a system which does not, is highly unlikely to hinge on whether the designers used aggregations or named associations. An aggregation can usually be replaced by an association called "has" without much loss of clarity.

Composition

It is not clear that composition is more expressive than a simple association labelled "contains". **Composition** expresses a whole-part relationship where the "parts" are definitely contained within the "whole".

Figure 5.6. The representation of composition in the UML
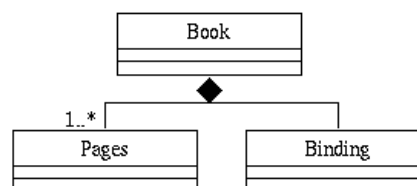
This example can be read as "a book consists of one or more pages, and one binding". Note the distinctions between this and the previous example. First, the binding and pages physically comprise

the book; it is impossible for the binding and pages to be in one place and book in another. Second, destruction of the book would imply destruction of the binding and pages.

Self-association and roles

So far we have shown how classes can enter into associations with other classes. However, it is allowable — and often essential — to model a class as being associated with itself. For example, suppose we are designing a system for managing contracts, and we want to show that a contract can consist of a number of small sub-contracts. Sub-contracts have exactly the same properties as contracts. We could represent this as follows:

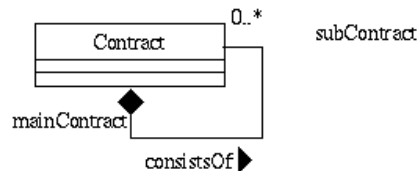This example says that one object of class "Contract" consists of zero or more objects of class "Contract". To clarify the relationship, it is customary to use role names. Where an association name describes the overall nature of the association, a role name is attached to the end of an association, and describes the role played by the object at that end. If the example above, one object plays the role of "main contract" while zero or more others play the role of "sub-contract". Because role names are so important in these cases, some CASE tools will report an error if they are omitted.

Link classes and link attributes

In many cases each instance of an association has properties of its own. It is sometimes helpful to show this in the model for increased expressiveness. In addition, it gives us a way to show that multiple classes are involved in the same association.

A link attribute simply assigns a value to a particular instance of the link, that is, for each pair of objects that are associated there is a particular value of the attribute. For example, in the top diagram of the figure shown below:

Figure 5.7. Link attributes
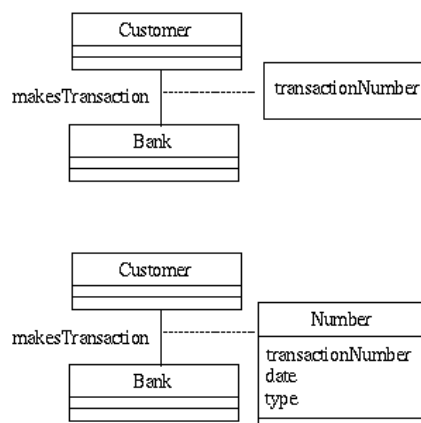
"transactionNumber" is a link attribute. This model says that the bank and its customers enter into transactions, and each transaction has a particular number.

However, the association may be too complex to be represented by a single attribute. In this case we can use a link class. With a link class there is one instance of the class for each pair of objects that

are associated. The link class can have any number of attributes, and can quite legitimately enter into relationships of its own with other classes in the system. In the example above, the link class "Number" defines each transaction carried out by the bank with its customers. It has attributes to indicate the nature and date of the transaction, as well as the number. Link classes can also have sub-classes. For example, it may be useful to indicate that there are different types of transaction by means of sub-classes, rather than simply having an attribute called "type".

Link classes are a very powerful and expressive tool, but do take some experience to able to use effectively.

Constraints and notes

Notes

As in use case modelling, it is often very helpful to be able to annotate a diagram to show particular features. The notation is exactly the same:

Figure 5.8. The notation for notes
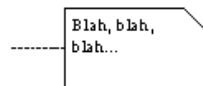
You can, and should, apply notes wherever you think they will be helpful to the reader, but they should not be viewed as a substitute for proper documentation. A class model is not complete until all classes, attributes, operations, and associations, have been documented. Notes are useful for pointing out specific details on diagrams.

Constraints

Constraints are additional information about a class that are not associated with any particular attribute or operation. A typical constraint might say, for example, that one attribute will have a value which depends on some other attribute. It is customary to write simple constraints below the class in braces, {like this}.

Constraints are becoming increasingly important in object-oriented modelling, as there is an increasing interest in formalising the technique. A scheme called the *object constraint language* is under development to support formalised constraints, but this is beyond the scope of this course.

Class-Responsibility-Collaborator cards

Class-responsibility-collaborator cards (CRC cards) are not a part of the UML specification, but they are a useful tool for organising classes during analysis and design.

A CRC card is a physical card representing a single class. Each card lists the class's name, attributes and methods (its *responsibilities*), and class associations (*collaborations*). The collection of these CRC cards is the *CRC model*.

Using CRC cards is a straightforward addition to object-oriented analysis and design:

1. Identify the classes.

2. List responsibilities.

3. List collaborators.

CRC cards can be used during analysis and design while classes are being discovered in order to keep track of them.

CRC cards have various benefits, which you might notice makes them very amenable to iterative and incremental process models, especially agile ones:

- They are **portable**: because CRC cards are physical objects, they can be used anywhere. Importantly, they can easily be used during group meetings.

- They are **tangible**: the participants in a meeting can all easily examine the cards, and hence examine the system.

- They have a **limited size**: because of their physicality, CRC cards can only hold a limited amount of information. This makes them useful to restricting object-oriented analysis and design from becoming too low-level.

Class *responsibilities* are the class's attributes and methods. Clearly, they represent the class's state and behaviour. *Collaborators* represent the associations the class has with other classes.

CRC cards are useful when the development of classes need to be divided between software engineers, as the cards can be physically handed over to them. A useful time to do this is when classes are being reviewed, for, say, determining whether they are appropriate in a design.

From model to program

During object-oriented development, the models will tend to become more detailed, and there will be a shift in standpoint from a logical to an implementation view. Developers using the waterfall or similar process models feel that there should be a strong correspondence between the final model and the program code it results in. However, iterative and agile methods often eschew heavy amounts of documentation, and realise that models (including object models) may not always accurately reflect the code, since both will always be changing as the code-base develops, and as the customers' requirements develop over time.

For agile methodologies it is important to remember that documentation is secondary to the software being produced. Slavishly attempting to specify every last component of the software in a detailed object model is antithetical to agile process methods, and there are many published studies showing that such complete, upfront designs are both inadequate (since requirements, and hence designs, continuously change) and detrimental to software engineering as a whole.

Dynamic behaviour

The object model which we have just been discussing is a *static* model of classes and their relationships. It does not show how the classes call on one another to perform the functions required in the software system.

These *dynamic* aspects of the object model are usually represented in other diagram types of the UML. We will be briefly discuss *interaction diagrams*.

Interaction diagrams

Interaction diagrams show how objects interact with one another. Specifically, they show which objects are currently executing, and what methods they are calling on other objects.

There are various kinds of interaction diagrams that can be employed in the UML. We will examine one specific type, the *sequence diagram*. Sequence diagrams show how objects interact with each other over *time*. In other words, sequence diagrams show the sequence of interactions between objects.

Figure 5.9. Sequence diagram notation in the UML
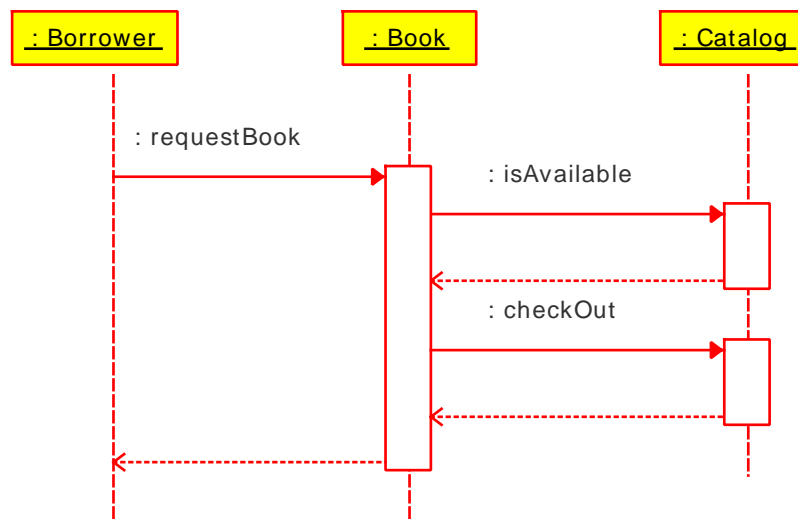
The figure above shows a sequence diagram of how books are checked out of a library. Notice how the objects are arranged horizontally along the top of the diagram, and are represented as rectangle boxes, just as classes are. It is important to notice the colon before the class name in these boxes — this colon tells us that the box represents not a class, but an instance of the class. If the instance should be given a specific name, then the name appears before the colon. For example, "artOfWar : Book" represents an object called "artOfWar", of class "Book". This is exactly the same as when specifying attributes in classes (see the section called "Attributes" above).

It is important to realise that it is because sequence diagrams represent the interactions a software system undergoes over time that it is interested in objects rather than classes. A running software system consists of instances of classes, and not of the classes themselves.

Time is represented vertically, with earlier events happening closer to the top. This means that sequence diagrams should be read from top to bottom. Each object also has a vertical **life-line** that shows us the object's period of existence. Interactions are represented by solid and dotted lines, each of which has an arrow and a label. Solid lines show *method calls*, and point from the life-line of the object making the call towards the object which will execute the method. The rectangles along the life-line represent the time over which an object is executing a function. Dotted lines represent a return of control from the object executing a method to and object which had originally called it.

Note that the method names should be operations belonging to the object to which the arrow is pointing. "Borrower" calls "requestBook", an operation belonging to an instance of class "Book".

A useful way of using sequence diagrams is to outline difficult use cases as a sequence diagram: this will highlight what objects are required in developing the use case, as well as what actions each object will perform in order to complete the use case. Further, a sequence diagram can then be thought of as showing how the system reacts to *events*, the events being the input from the actor in the use case.

The UML also prompts us to use sequence diagrams in this way: remember that actors in use cases are themselves classes. We can easily use instance of these classes (of these actors) as valid objects in sequence diagrams.

Summary

Class modelling is the key modelling technique for object-oriented designers. If you are in a position to say that you understand class modelling, then you can reasonably claim that you understand object-oriented design. Of course, an understanding of other object-oriented techniques (e.g., use case modelling) will also be helpful.

Class modelling centres on a small number of philosophical points, which must be understood extremely well. There is not a great deal of factual material to remember, but these key points are crucial. These are the most important issues:

- The distinction between objects and classes

- The nature of attributes and operations

- The use and specification of associations, especially generalisation

In constructing a class model, the first step must be to obtain a working list of candidate classes. As design progresses, these classes will be refined and some will be deleted. At an even later stage, new classes will be introduced to support more detailed, implementation-related concepts.

A class model is refined by finding and specifying associations. Doing this helps to determine whether the initial choice of classes is a sound one, and how to improve it.

Ultimately the class model will be transformed into a computer program.

Review

Questions

Review Question 1

Which of the following entities are likely to be of interest when considering system design from a "logical" point of view, and which from an "implementation" point of view? Which are relevant to all standpoints? Clerk; Screen; Mouse (the input device, not the rodent); File; List; Ledger; Service technician; Database; Customer;

A discussion of this question can be found at the end of this chapter.

Review Question 2

Fill in the gaps with an appropriate word or phrase, or select the correct one of the indicated choices:

A class encapsulates the behaviour and properties of a number of [_____]. The members of a class are also referred to as [_____]. The minimum number of instances of a class is [_____], the maximum is [_____]. In object-oriented design, an object [can/cannot] be be a instance of more than one class, and an object [can/cannot] change which class it is an instance of.

A discussion of this question can be found at the end of this chapter.

Review Question 3

Suppose you are designing a computer system to carry out stock control in a large book warehouse. The system should store extensive details of each book, including at least the following information: authors, title, publisher, date of publication, size, number of pages, binding format, shelf location, re-order level, purchase price, sale price. Which of these properties of books do you think are relevant to the *conceptual* class "Book", and what properties to the *concrete* class "Book"? This problem is somewhat more difficult than it first appears; see the answer for further discussion.

A discussion of this question found at the end of this chapter.

Review Question 4

Fill in the blanks with an appropriate word or phrase:

Attributes define the properties of a class. In a class specification an attribute has a [_____] and a [_____] (e.g., number, text). In each object of that class the attribute will also have a

[_____]. This may be different for each object, but no instance of the class will lack the attribute entirely. For example if I say that a particular vehicle has 12 wheels, I am implying that vehicles have a attribute whose name is "[_____]", whose type is "[_____]" and whose value in the particular case is [12]. The minimum number of attributes that a class may have is [_____]. The maximum number is unlimited, but in practice it is difficult to manage a class with more than about 30 attributes. Java programmers tend to use the term [_____] in preference to attribute, but these are essentially the same thing.

A discussion of this question can be found at the end of this chapter.

Review Question 5

Draw a UML class symbol that has the following specification. The class is called "Customer". It has four attributes: name (which is text), address (also text), balance (a number), and credit rating (type currently unknown). It has one operation: "printStatement()", which has no parameters or return value.

A discussion of this question can be found at the end of this chapter.

Review Question 6

In the early stages of development it is usually better to identify too many classes than too few. You can then rationalise you choice of classes by removing, modifying and merging them. The following list is of classes that may be considered for deletion or absorption into a new class. Suggest why this might be the case for each example.

- Fred Bloggs, a customer of the bank

- Colour

- Hard disk

- Central processing unit

- Linked list

- Red

- Test procedure

- Print

- List of customers

For as many of these examples as you can, suggest an application where it is not a bad choice of class.

A discussion of this question can be found at the end of this chapter.

Review Question 7

Draw the class diagram for the radio/televsion/car example given in the text, showing where multiple inheritance occurs.

A discussion of this question can be found at the end of this chapter.

Review Question 8

Construct a class model to represent the following information. The customer is a company that specialise in the supply of specialist musical recordings, of the type that are difficult to obtain through mainstream record shops. The customer would like to make it possible for their clients to order their products on-line using a Web browser. The on-line system is to provide full details of each musical recording, and will be integrated into an automated stock control system. In this way their client will

be able to tell immediately if the items required are in stock. The customers describes the details of their stock as follows:

> We stock about 10,000 different musical recordings, usually with 1-10 of each item in stock at any given time. With more popular items we will stock ten or more copies of the same recording, perhaps in different formats.

> We stock recordings in different formats: cassette tapes, audio CDs and audio DVDs. The publishers we buy from will normally supply a recording in more than one format. For example, usually we will get CDs and cassettes of a given recording from the same publisher.

> We buy our stock from music publishers. For each publisher we record the name and address, and the name and e-mail address of our contact person there. When we place an order we need to know the publisher's catalogue number, which is generally different for different formats.

> When a customer looks at our Web site, we want to be able to give extensive details for all our recordings. For example, we will want to display the title, performer, composer, date and venue of recording, copyright holders and some general information. In addition, we want to display the titles and durations of all the musical tracks. This is complicated by the fact that the CD, DVD and cassette formats generally don't have the same tracks. Because a DVD is longer than a CD, it will usually have a few extra tracks. A cassette tape usually has even fewer tracks than the CD.

A discussion of this question can be found at the end of this chapter.

Review Question 9

Redraw the contract-subcontract relationship shown below so that it does not use a self-association. Do you think your model is more or less expressive in this form?
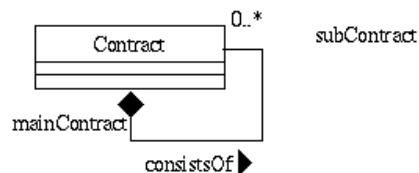
A discussion of this question can be found at the end of this chapter.

Review Question 10

What is meant by the statement "objects are not classes, but instances of classes. However, classes can be considered objects"? In other words, in what sense is a class also an object? (This is a tricky question if you are not absolutely clear on the distinction between "classes" and "objects").

A discussion of this question can be found at the end of this chapter.

Review Question 11

Write a simple, one-line definition of each of the following key terms. Try to do this without referring back to the notes if possible. For each term, give an example.

- Attribute

- Operation

- Self-association

- Generalisation

- Polymorphism

- Link class

- Multiplicity

- Stereotype

A discussion of this question can be found at the end of this chapter.

Review Question 12

Which of the following are true? Give an example to support your decision.

- If Z is a subclass of Y, and Y is a subclass of X, then Z is a subclass of X

- If Z is associated with Y by an association A, and Y is associated with X by an association A, then Z is associated with X by an association A

- If Z is associated with Y by an association A and by an association B, then A and B are different names for the same association

- If each object of class Z is associated with ten objects of class Y, and each object of class Y is associated with ten objects of class X, then there are 100 objects of class X for each object of class Z

A discussion of this question can be found at the end of this chapter.

Review Question 13

What is the difference between class modelling, and entity-relationship modelling? In what ways are classes and entities similar? In what ways are they different? (If you have not completed the part of the course that deals with entity-relationship modelling, you may find this question difficult).

A discussion of this question can be found at the end of this chapter.

Review Question 14

Consider these three classes: Rectangle, Circle, and Line, which are subclasses of a general Shape class. These classes have operations drawRectangle(), drawCircle(), and drawLine() respectively. Each of these methods cause the appropriate shape to be drawn. Is this system exhibiting polymorphism or not?

A discussion of this question can be found at the end of this chapter.

Review Question 15

What is the difference between a link class and a link attribute? Gives examples of each.

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Probably "Clerk", "Customer", "Ledger" and "Service technician" are important in a logical standpoint. None of these are likely to transform directly into parts of a computer program. On the other hand, "File", "List" and "Database" probably will. "Screen" and "Mouse" are less easy to categorise. In some cases these will in fact simply be alternative representations of the human beings that will interact with a system. For example, a clerk may use a keyboard and a mouse; in some systems it is

the fact that these particular devices are in use that is important, but in most it will be the person and that person's role that are important.

Discussion of Review Question 2

A class encapsulates the behaviour and properties of a number of [objects, entities]. The members of a class are also referred to as [instances]. The minimum number of instances of a class is [zero], the maximum is [unlimited, infinite]. In object-oriented design, an object [cannot] be be a instance of more than one class, and an object [cannot] change which class it is an instance of.

Discussion of Review Question 3

Some things are clearly properties of the conceptual class. For a book, things like the title, author, date of publication and publisher are probably conceptual. They will be independent of any physical realisation of that book. For example, there are many copies of Tolstoy's "War and Peace" in existence, in many different formats, but all share the same fundamental properties.

In a library, the shelf location and bar-code number are properties of the physical book.

Less straightforward cases are properties like the format (size, shape, binding) of the book. These are clearly "concrete" properties. However, there are many different copies of each of these formats. Are these individual copies "physical" and the formats themselves "conceptual"? This is not a straightforward question. It may be in a very complex book handling system that we have to institute a number of different classes to represent book information.

Discussion of Review Question 4

Attributes define the properties of a class. In a class specification an attribute has a [name] and a [type] (e.g., number, text). In each object of that class the attribute will also have a [value]. This may be different for each object, but no instance of the class will lack the attribute entirely. For example if I say that a particular vehicle has 12 wheels, I am implying that vehicles have a attribute whose name is "[number of wheels]", whose type is "[number, integer]" and whose value in the particular case is [12]. The minimum number of attributes that a class may have is [zero]. The maximum number is unlimited, but in practice it is difficult to manage a class with more than about 30 attributes. Java programmers tend to use the term [instance variable] in preference to attribute, but these are essentially the same thing.

Discussion of Review Question 5

Discussion of Review Question 6

Bad classes:

- Fred Bloggs, a customer of the bank: this is potentially a bad class because Fred Bloggs is an object (instance), perhaps of class Customer. It is unlikely that there will be a whole class of Fred Bloggses.

- Colour: colour is likely to be an attribute of something. If you have studied Java programming you will probably come across a class called Colour. However, Java does this as an extension of

the language, not because Colour represents anything in a model. However, in a program for, for example, colour mixing in printing machinery, perhaps Colour is important enough to be a class in its own right.

- Hard disk: this is a bad class because it is concerned with the internal operation of the computer. In modelling, we want to avoid detail of this depth. "Document" and perhaps "Folder" may be reasonable modelling concepts. However, in an application to design computers, HardDisk may be a sensible class.

- Central processing unit: has all the same problems as the example above

- Linked list: if you didn't know what a linked list was, then that is as good a reason as any for getting rid of it. Even if you did, it is still too technical to use in modelling

- Red: is probably the value of an attribute

- Test procedure: this is probably an operation, or perhaps a use case

- Print: this is almost certainly an operation (or a Printer class perhaps?)

- List of customers: too much detail. "Customer" is a sensible class name. The fact that there are more than one customer should be represented by the multiplicities of associations with other classes (see later).

Discussion of Review Question 7
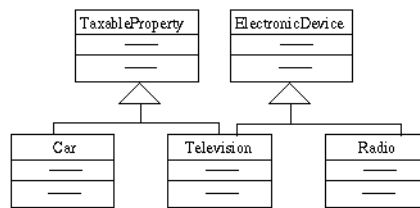
This is a somewhat contrived example, and is unlikely to occur in a real application. However, multiple inheritance does occur in real applications all the time. It is probably more important in programming, rather than modelling as such. For example, a popular style of programming called "mix-in" programming makes extensive use of multiple inheritance. In mix-in programming a large number of very simple, general-purpose classes are defined, and these are inherited in various combinations by the major classes. Because something is important in programming, this does not necessarily mean it should be used in modelling. This topic is an important and controversial one.

Discussion of Review Question 8

There is no single right answer to this question. You should be prepared to discuss you solution with your tutor/classmates (hint: a good answer will probably contain between 5 and 10 classes).

Discussion of Review Question 9
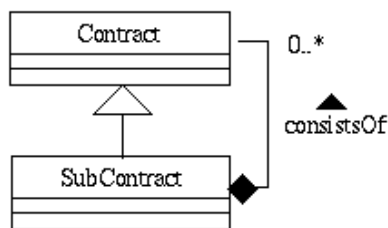
In this alternative formulation, there is no self-association, but a SubContract is shown as a sub-class of Contract. The representation that used a self-association did not make this class/sub-class relationship explicit; indeed it may not really be true. If one is prepared to accept the implicit assumption that SubContract is a sub-class of Contract, then the diagram shown above does effectively remove the self-association. This may be a reasonable approach for someone who really find self-associations difficult to follow. So perhaps some improvement in expressiveness has been gained, but at the expense of a slight loss of accuracy.

Discussion of Review Question 10

Suppose a class model has three classes, called ClassA, ClassB and ClassC. Objects of these classes will share properties (this is the definition of being an instance of a class, after all). However, the classes themselves will also have properties in common. For example, they will all the attribute "name". Its value is "ClassA" for ClassA, "ClassB" for ClassB, etc. All classes will have the property of being able to enter into relationships with other classes. All classes will have the attribute "numberOfAttributes". Thus we can say that there is a class called "Class", which embodies the behaviour of all classes. Each actual class is an instance of class "Class".

This is not simply an academic point. Being able to express a modelling notation in terms of the modelling notation itself is a very powerful way to ensure that the modelling system is consistent. If you study the UML specification documents, you will find that all UML diagram rules are themselves specified in UML notation. This is called meta-modelling

Discussion of Review Question 11

- Attribute: a property that objects of a class have, that can be expressed in terms of a name, a type and a value. For example, class "Vehicle" has attribute "numberOfWheels". Different vehicles have different values of this attribute.

- Operation: a well-defined unit of behaviour of an object of a class. For example, class Vehicle has operation "startEngine()".

- Self-association: an association between one or more objects of a class, and one or more objects of the same class. For example, each object of class "LivingThing" is involved in a one-to-many association called "eats" with other objects of the same class (technically I suppose that some living things don't eat other living things, but I hope you get the general idea).

- Generalisation: the property that a class has of representing behaviour and attributes of a number of subclasses. For example, class Vehicle generalises classes Car, Bus and Tram, etc.

- Polymorphism: differences in behaviour between different sub-classes of the same base class. For example, all sub-classes of Vehicle have the operation "startEngine()", but the mechanism of starting the engine is different in each case.

- Link class: a class that specifies the details of an association between two other objects. For example, the relationship between a seller and a buyer may be complex enough to need a class (e.g., SalesTransaction) to represent its details.

- Multiplicity: the number of objects that take part in each end of an association. For example, the multiplicity of the relationship between Driver and Vehicle is 1 to (1-*), meaning a driver can drive one or more vehicles.

- Stereotype: a category to which a class belongs, e.g., "Actor", or "User interface element".

Discussion of Review Question 12

- *If Z is a subclass of Y, and Y is a subclass of X, then Z is a subclass of X*. This is correct, and a fundamental principle of object orientation. If mammal is a type of animal, and dog is a type of mammal, then dog is a type of animal. Can you think of a dog that is a mammal but not an animal?

- *If Z is associated with Y by an association A, and Y is associated with X by an association A, then Z is associated with X by an association A.* This is false. For example, lions eat zebra and zebra eat grass. But lions don't eat grass. The association is not carried across the classes. This is one of the things that distinguishes generalisation/specialisation from other relationships.

- *If Z is associated with Y by an association A and by an association B, then A and B are different names for the same association.* This is also false. Two objects can be associated in completely different ways. For example, customers deposit money in a bank account; customers withdrawn money from a bank account. "Deposit" and "withdraw" are two totally different associations.

- *If each object of class Z is associated with ten objects of class Y, and each object of class Y is associated with ten objects of class X, then there are 100 objects of class X for each object of class Z.* This is correct. For example, if a program has ten screen windows, and each window has ten buttons, then there will be 100 buttons for each program.

Discussion of Review Question 13

Class diagrams can be considered to be highly-developed and regularised entity relationship models (ERMs). Class modelling provides a definite notation and meaning for concepts that are weakly-defined in ERMs. For example, an relationship called "is-a" can be used in an ERM to show that one thing is a type of another thing, or that one thing is an instance of another thing. Moreover, when an "is-a" relationship is used, it does not necessarily denote any inheritance of properties. In ERMs properties are usually represented as entities as well, so this would be difficult to show. The same applies to relationships that are often written as "has-a". This is an aggregation, but the different types of aggregation are not as well defined in ERM as they are in class modelling.

For all these differences, ERMs and class diagrams are more similar than they are different; both attempt to show the static structure of system in terms of the "things" (entities; classes) of which it is comprised.

Discussion of Review Question 14

The system as described is not polymorphic. There is a difference in behaviour between the different subclasses of Shape, and this difference is in an operation that is philosophically common to the three classes. That is, drawCircle(), drawRectangle(), and drawLine() are obviously drawing operations. However, there is no corresponding "drawing" operation in the base class "Shape". The system could not invoke the drawing operation in Shape, with a view to the correct sub-class being drawn. To be polymorphic, the base class needs to have an operation called "draw()" that is abstract, that is, does nothing. The sub-classes all provide their own implementations of "draw()". Now if there is an object of an sub-class of Shape (it doesn't matter which), another object can cause it to be drawn by invoking the "draw()" operation. The presence of "draw" as an abstract operation in Shape guarantees that all sub-classes of shape will respond to a "draw" operation. This is a powerful simplifying principle.

Discussion of Review Question 15

Both link classes and link attributes provide additional information about associations than can be carried by a simple association name. A link attribute carries only a single piece of information and, being and attribute, cannot interact directly with other objects. A link class is a fully-fledged class and can take part in class relationships of its own. It can carry as much information as required. Link attributes are simpler to read and simpler to implement, but are not as versatile. For example, an employer and an employee may interact in a way defined by a contract. A simple link attribute might be a contract number. This would identify different contractual agreements between the same entities. However, implementing Contract as a class would allow a lot of extra information to be represented. Of course, it is only worth doing this if it improves the expressiveness or accuracy of the model.

Chapter 6. Data-Flow Diagrams

Objectives

At the end of this chapter you should be able to:

• Explain the purpose of data-flow diagrams.

• Describe the meaning of the symbols used in data-flow diagrams.

• Describe the generic framework activities at which data flow diagrams can be used and the corresponding roles of data-flow diagrams in these stages.

• Construct simple data-flow diagrams from a textual description.

• Construct a levelled set of data-flow diagrams.

• Understand how to check the consistency of related data-flow diagrams.

Introduction to data-flow diagrams

What are data-flow diagrams?

Data-flow diagrams (DFDs) model a perspective of the system that is most readily understood by users – the flow of information through the system and the activities that process this information.

Data-flow diagrams provide a graphical representation of the system that aims to be accessible to computer specialist and non-specialist users alike. The models enable software engineers, customers and users to work together effectively during the analysis and specification of requirements. Although this means that our customers are required to understand the modeling techniques and constructs, in data-flow modeling only a limited set of constructs are used, and the rules applied are designed to be simple and easy to follow. These same rules and constructs apply to all data-flow diagrams (i.e., for each of the different software process activities in which DFDs can be used).

An example data-flow diagram

An example of part of a data-flow diagram is given below. Do not worry about which parts of what system this diagram is describing – look at the diagram to get a feel for the symbols and notation of a data-flow diagram.

Figure 6.1. An example data-flow diagram
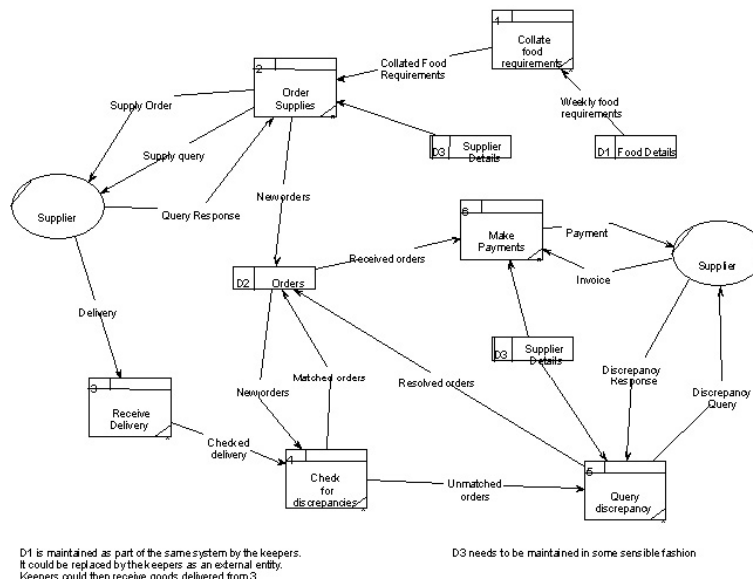
As can be seen, the DFD notation consists of only four main symbols:

1. Processes — the activities carried out by the system which use and transform information. Processes are notated as rectangles with three parts, such as "Order Supplies" and "Make Payments" in the above example.

2. Data-flows — the data inputs to and outputs from to these activities. Data-flows are notated as named arrows, such as "Delivery" and "Supply Order" in the example above.

3. External entities — the sources from which information flows into the system and the recipients of information leaving the system. External entities are notated as ovals, such as "Supplier" in the example above.

4. Data stores — where information is stored within the system. Data stores are notated as rectangles with two parts, such as "Supplier Details" and "Orders" in the example above.

The diagrams are supplemented by supporting documentation including a **data dictionary**, describing the contents of data-flows and data stores; and **process definitions**, which provide detailed descriptions of the processes identified in the data-flow diagram.

The benefits of data-flow diagrams

Data-flow diagrams provide a very important tool for software engineering, for a number of reasons:

•   The system scope and boundaries are clearly indicated on the diagrams (more will be described about the boundaries of systems and each DFD later in this chapter).

•   The technique of decomposition of high level data-flow diagrams to a set of more detailed diagrams, provides an overall view of the complete system, as well as a more detailed breakdown and description of individual activities, where this is appropriate, for clarification and understanding.

Note

Use-case diagrams also provide a partition of a software-system into those things which are inside the system and those things which are outside of the system.

Case study

We shall be using the following case study to explore different aspects of data-flow modeling and diagrams.

Video-Rental LTD case study

Video-Rental LTD is a small video rental store. The store lends videos to customers for a fee, and purchases its videos from a local supplier.

A customer wishing to borrow a video provides the empty box of the video they desire, their membership card, and payment – payment is always with the credit card used to open the customer account. The customer then returns the video to the store after watching it.

If a loaned video is overdue by a day the customer's credit card is charged, and a reminder letter is sent to them. Each day after that a further card is made, and each week a reminder letter is sent. This continues until either the customer returns the video, or the charges are equal to the cost of replacing the video.

New customers fill out a form with their personal details and credit card details, and the counter staff give the new customer a membership card. Each new customer's form is added to the customer file.

The local video supplier sends a list of available titles to Video-Rental LTD, who decide whether to send them an order and payment. If an order is sent then the supplier sends the requested videos to the store. For each new video a new stock form is completed and placed in the stock file.

The different kinds (and levels) of data-flow diagrams

Although all data-flow diagrams are composed of the same types of symbols, and the validation rules are the same for all DFDs, there are three main types of data-flow diagram:

- **Context diagrams** — context diagram DFDs are diagrams that present an overview of the system and its interaction with the rest of the "world".

- **Level 1 data-flow diagrams** — Level 1 DFDs present a more detailed view of the system than context diagrams, by showing the main sub-processes and stores of data that make up the system as a whole.

- **Level 2 (and lower) data-flow diagrams** — a major advantage of the data-flow modelling technique is that, through a technique called "levelling", the detailed complexity of real world systems can be managed and modeled in a hierarchy of abstractions. Certain elements of any data-flow diagram can be decomposed ("exploded") into a more detailed model a level lower in the hierarchy.

During this unit we shall investigate each of the three types of diagram in the sequence they are described above. This is both a sequence of increasing complexity and sophistication, and also the sequence of DFDs that is usually followed when modeling systems.

For each type of diagram we shall first investigate *what* the features of the diagram are, then we shall investigate *how* to create that type of diagram. However, before looking at particular kinds of data-flow diagrams, we shall briefly examine each of the symbols from which DFDs are composed.

Elements of data-flow diagrams

Four basic elements are used to construct data-flow diagrams:

- processes

- data-flows

- data stores

- external entities

The rest of this section describes each of the four elements of DFDs, in terms of their purpose, how the element is notated and the rules associated with how the element relates to others in a diagram.

Notation and software

A number of different notations exist for depicting these elements, although it is only the shape of the symbols which vary in each case, not the underlying logic. This unit uses the *Select SSADM* notation in the description and construction of data-flow diagrams.

As data-flow diagrams are not a part of the UML specification, *ArgoUML* and *Umbrello* do not support their creation. However, *Dia* is free software available for both *Windows* and *Ubuntu* which does support data-flow diagrams.
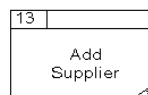
Processes

Purpose

Processes are the essential activities, carried out within the system boundary, that use information. A process is represented in the model only where the information which provides the input into the activity is manipulated or transformed in some way, so that the data-flowing out of the process is changed compared to that which flowed in.

The activity may involve capturing information about something that the organisation is interested in, such as a customer or a customer's maintenance call. It may be concerned with recording changes to this information, a change in a customer's address for example. It may require calculations to be carried out, such as the quantity left in stock following the allocation of stock items to a customer's job; or it may involve validating information, such as checking that faulty equipment is covered by a maintenance contract.

Notation

Processes are depicted with a box, divided into three parts.

Figure 6.2. The notation for a process

The top left-hand box contains the process number. This is simply for identification and reference purposes, and does not in any way imply priority and sequence.

The main part of the box is used to describe the process itself, giving the processing performed on the data it receives.

The smaller rectangular box at the bottom of the process is used in the *Current Physical Data-Flow Diagram* to indicate the location where the processing takes place. This may be the physical location — the *Customer Services Department* or the *Stock Room*, for example. However, it is more often used to denote the staff role responsible for performing the process. For example, *Customer Services*, *Purchasing*, *Sales Support*, and so on.

Rules

The rules for processes are:

- Process names should be an imperative verb specific to the activity in question, followed by a pithy and meaningful description of the object of the activity. *Create Contract*, or *Schedule Jobs*, as opposed to using very general or non-specific verbs, such as *Update Customer Details* or *Process Customer Call*.

- Processes may not act as data sources or sinks. Data flowing into a process must have some corresponding output, which is directly related to it. Similarly, data-flowing out of a process must have some corresponding input to which it is directly related.

- Normally only processes that transform system data are shown on data-flow diagrams. Only where an enquiry is central to the system is it included.

- Where a process is changing data from a data store, only the changed information flow to the data store (and not the initial retrieval from the data store) is shown on the diagram.

- Where a process is passing information from a data store to an external entity or another process, only the flow from the data store to the process is shown on the diagram.

Data-flows

Purpose

A data-flow represents a package of information flowing between two objects in the data-flow diagram. Data-flows are used to model the flow of information into the system, out of the system, and between elements within the system.

Occasionally, a data-flow is used to illustrate information flows between two external entities, which is, strictly speaking, outside of the system boundaries. However, knowledge of the transfer of information between external entities can sometimes aid understanding of the system under investigation, in which case it should be depicted on the diagram.

Notation

A data-flow is depicted on the diagram as a directed line drawn between the source and recipient of the data-flow, with the arrow depicting the direction of flow.

Figure 6.3. Notation for a data-flow

Supplier details
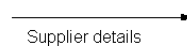
The directed line is labelled with the data-flow name, which briefly describes the information contained in the flow. This could be a *Maintenance Contract*, *Service Call Details*, *Purchase Order*, and so on.

Data-flows between external entities are depicted by dashed, rather than unbroken, lines.

Rules

The rules for drawing data-flows are:

- Information always flows to or from a process; the other end of the flow may be an external entity, a data store or another process. An occasional exception to this rule is a data-flow between two external entities.

- Data stores may not be directly linked by data-flows; information is transformed from one stored state to another via a process.

- Information may not flow directly from a data store to an external entity, nor may it flow from an external entity directly to a data store. This communication and receipt of information stored in the system always takes place via a process.

- The sources (where data of interest to the system is generated without any corresponding input) and sinks (where data is swallowed up without any corresponding output) of data-flows are always represented by external entities.

- When something significant happens to a data-flow, as a result of a process acting on it, the label of the resulting data-flow should reflect its transformed status. For example, "Telephoned Service Call" becomes "Service Call Form" once it has been logged.

Data stores

Purpose

A data store is a place where data is stored and retrieved within the system. This may be a file, *Customer Contracts* file for example, a catalogue or reference list, *Options Lists* for example, a log book such as the *Job Book*, and so on.

Notation

A data store is represented in the data-flow diagram by a long rectangle, containing two locations.

Figure 6.4. Notation for a data store
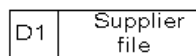
The small left-hand box is used for the identifier, which comprises a numerical reference prefixed by a letter.

The main area of the rectangle is labelled with the name of the data store. Brief names are chosen to reflect the content of the data store.

Rules

The rules for representing data stores are:

- One convention that could be used is to determine the letter identifying a data store by the store's nature.

- "M" is used where a manual data store is being depicted.

- "D" is used where it is a computer based data store.

- "T" is used where a temporary data store is being represented.

- Data stores may not act as data sources or sinks. Data flowing into a data store must have some corresponding output, and vice versa.

- Because of their function in the storage and retrieval of data, data stores often provide input data-flows to receive output flows from a number of processes. For the sake of clarity and to avoid crisscrossing of data-flows in the data-flow diagram, a single data store may be included in the diagram at more than one point. Where the depiction of a data store is repeated in this way, this is signified by drawing a second vertical line along the left-hand edge of the rectangle for each occurrence of the data store.

External entities

Purpose

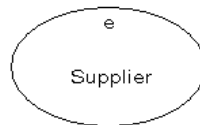External entities are entities outside of the system boundary which interact with the system, in that they send information into the system or receive information from it. External entities may be external to the whole organisation — as in *Customer* and *Supplier* in our running example; or just external to the application area where users' activities are not directly supported by the system under investigation. *Accounts* and *Engineering* are shown as external entities as they are recipients of information from the system. *Sales* also provide input to the system.

External entities are often referred to as *sources* and *sinks*. All information represented within the system is sourced initially from an external entity. Data can leave the system only via an external entity.

Notation

External entities are represented on the diagram as ovals drawn outside of the system boundary, containing the entity name and an identifier.

Figure 6.5. Notation for external entities

Names consist of a singular noun describing the role of the entity. Above the label, a lower case letter is used as the identifier for reference purposes.

Rules

The rules associated with external entities are:

• Each external entity must communicate with the system in some way, thus there is always a data-flow between an external entity and a process within the system.

• External entities may provide and receive data from a number of processes. It may be appropriate, for the sake of clarity and to avoid crisscrossing of data flows, to depict the same external entity at a number of points on the diagram. Where this is the case, a line is drawn across the left corner of the ellipse, for each occurrence of the external entity on the diagram. *Customer* is duplicated in this way in our example.

Multiple copies of entities and data stores on the same diagram

At times a diagram can be made much clearer by placing more than one copy of an external entity or data store in different places — this can avoid a tangle of crossing data-flows.

Where more than one copy of an external entity appears on a diagram it has a cut off corner in the top left, such as below:

Figure 6.6. How to notate duplicated external entities
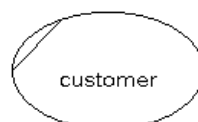
When more than one copy of a data store appears on a diagram it has a cut off left-side, such as below:

Figure 6.7. How to notate duplicate data stores
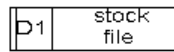
Context diagrams

What is a context diagram?

The context diagram is used to establish the context and boundaries of the system to be modelled: which things are inside and outside of the system being modelled, and what is the relationship of the system with these external entities.

A context diagram, sometimes called a *level 0 data-flow diagram*, is drawn in order to define and clarify the boundaries of the software system. It identifies the flows of information between the system and external entities. The entire software system is shown as a single process.

A possible context diagram for the Video-Rental LTD case study is shown below.

Figure 6.8. A context diagram for Video-Rental LTD

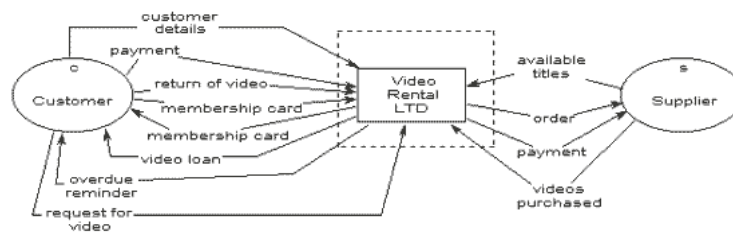The process of establishing the analysis framework by drawing and reviewing the context diagram inevitably involves some initial discussions with users regarding problems with the existing system and the specific requirements for the new system. These are formally documented along with any specific system requirements identified in previous studies.

Having agreed on the framework, the detailed investigation of the current system must be planned. This involves identifying how each of the areas included within the scope will be investigated. This could be by interviewing users, providing questionnaires to users or clients, studying existing system documentation and procedures, observation and so on. Key users are identified and their specific roles in the investigation are agreed upon.

Constructing a context diagram

In order to produce the context diagram and agree on system scope, the following must be identified:

• external entities

• data-flows

You may find the following steps useful:

1. Identify data-flows by listing the major documents and information flows associated with the system, including forms, documents, reference material, and other structured and unstructured information (emails, telephone conversations, information from external systems, etc.).

2. Identify external entities by identifying sources and recipients of the data-flows, which lie outside of the system under investigation. The actors an any use case models you have created may often be external entities.

3. Draw and label a process box representing the entire system.

4. Draw and label the external entities around the outside of the process box.

5. Add the data-flows between the external entities and the system box. Where documents and other packets of information flow entirely within the system, these should be ignored from the point of view of the context diagram – at this stage they are hidden within the process box.

This system boundary and details depicted in the context diagram should then be discussed (and updated if necessary) in consultation with your customers until an agreement is reached.

Having defined the system boundary and scope, the areas for investigation will be determined, and appropriate techniques for investigating each area will need to be decided.

Level 1 data-flow diagrams

What is a level 1 DFD?

As described previously, context diagrams (level 0 DFDs) are diagrams where the whole system is represented as a single process. A level 1 DFD notates each of the main sub-processes that together form the complete system. We can think of a level 1 DFD as an "exploded view" of the context diagram.

A possible level 1 DFD for the Video-Rental LTD case study is as follows:

Figure 6.9. A level 1 DFD for Video-Rental LTD
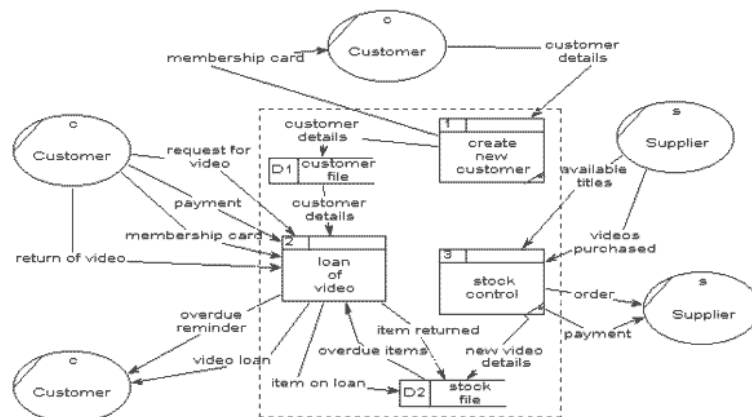
Notice that the external entities have been included on this diagram, but outside of the rectangle that represents the boundary of this diagram (i.e., the system boundary). It is not necessary to always show the external entities on level 1 (or lower) DFDs, however you may wish to do so to aid clarity and understanding.

We can see that on this level 1 DFD there are a number of data stores, and data-flows between processes and the data stores.

It is important to notice that the same data-flows to and from the external entities appear on this level 1 diagram and the level 0 context diagram. Each time a process is expanded to a lower level, the lower level diagram must show all the same data-flows into, and out of the higher level process it expands.

Constructing level 1 DFDs

If no context diagram exists, first create one before attempting to construct the level 1 DFD (or construct the context diagram and level 1 DFD simultaneously).

The following steps are suggested to aid the construction of Level 1 DFD:

1. Identify processes. Each data-flow into the system must be received by a process. For each data-flow into the system examine the documentation about the system and talk to the users to establish a plausible process of the system that receives the data-flow. Each process must have at least one output data-flow. Each output data-flow of the system must have been sent by a process; identify the processes that sends each system output.

2. Draw the data-flows between the external entities and processes.

3. Identify data stores by establishing where documents / data needs to be held within the system. Add the data stores to the diagram, labelling them with their local name or description.

4. Add data-flows flowing between processes and data stores within the system. Each data store must have at least one input data-flow and one output data-flow (otherwise data may be stored, and never used, or a store of data must have come from nowhere). Ensure every data store has input and output data-flows to system processes. Most processes are normally associated with at least one data store.

5. Check diagram. Each process should have an input and an output. Each data store should have an input and an output. Check the system details so see if any process appears to be happening for no reason (i.e., some "trigger" data-flow is missing, that would make the process happen).

Decomposing diagrams into level 2 and lower hierarchical levels

What is a level 2 (or lower) DFD?

We have already seen how a level 0 context diagram can be decomposed (exploded) into a level 1 DFD. In DFD modeling terms we talk of the context diagram as the "parent" and the level 1 diagram as the "child".

This same process can be applied to each process appearing within a level 1 DFD. A DFD that represents a decomposed level 1 DFD process is called a level 2 DFD. There can be a level 2 DFD for each process that appears in the level 1 DFD.

A possible level 2 DFD for process "2: Loan of video" of the level 1 DFD is as follows:

Figure 6.10. A level 2 data-flow diagram for Video-Rental LTD
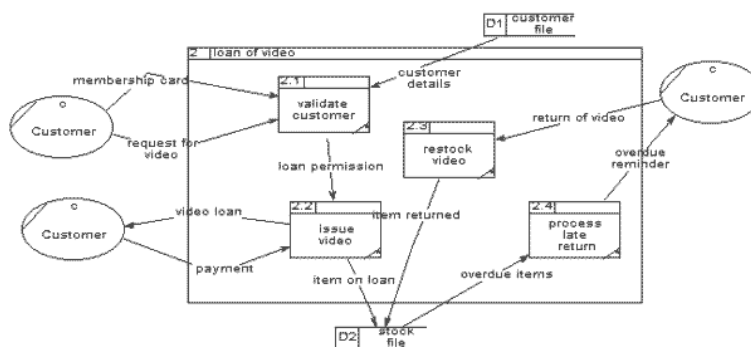
Note, that every data-flow into and out of the parent process must appear as part of the child DFD. The numbering of processes in the child DFD is derived from the number of the parent process – so all processes in the child DFD of process 2, will be called 2.X (where X is the arbitrary number of the process on the level 2 DFD). Also there are no new data-flows into or out of this diagram – this kind of data-flow validation is called balancing.

Look at the rectangular boundary for this level 2 DFD. Outside the boundary is the external entity "Customer". Also outside the boundary are the two data stores – although these data stores are inside the system (see the level 1 DFD), they are outside the scope of this level 2 DFD.

Constructing level 2 (and lower) DFDs — functional decomposition

The level 1 data-flow diagram provides an overview of the system. As the software engineers' understanding of the system increases it becomes necessary to expand most of the level 1 processes to a second or even third level in order to depict the detail within it. Decomposition is applied to each process on the level 1 diagram for which there is enough detail hidden within the process. Each process on the level 2 diagrams also needs to be checked for possible decomposition, and so on.

A process box that cannot be decomposed further is marked with an asterisk in the bottom right hand corner. A brief narrative description of each bottom-level process should be provided with the data-flow diagrams to complete the documentation of the data-flow model. These make up part of the **process definitions** which should be supplied with the DFD.

Each process on the level 1 diagram is investigated in more detail, to give a greater understanding of the activities and data-flows. Normally processes are decomposed where:

- There are more than six data-flows around the process

- The process name is complex or very general which indicates that it incorporates a number of activities.

The following steps are suggested to aid the decomposition of a process from one DFD to a lower level DFD. As you can see they are very similar to the steps for creating a level 1 DFD from a context diagram:

1. **Make the process box on the level 1 diagram the system boundary on the level 2 diagram that decomposes it**. This level 2 diagram must balance with its "parent" process box — the data-flows to and from the process on the level 1 diagram will all become data-flows across the system boundary on the level 2 diagram. The sources and recipients of data-flows across the level 2 system boundary are drawn outside the boundary and labelled exactly as they are on the level 1 diagram. Note that these sources and recipients may be data stores as well external entities or other processes — this is because a data store in a level 1 diagram will be outside the boundary of a level 2 process that sends or receives data-flows to/from the data store.

2. **Identify the processes inside the level 2 system boundary and draw these processes and their data-flows**. Remember, each data-flow into and out of the level 2 system boundary should be to/from a process. Using the results of the more detailed investigation, filter out and draw the processes at the lower level that send and receive information both across and within the level 2 system boundary. Use the level numbering system to number sub-processes so that, for example, process 4 on the level 1 diagram is decomposed to sub-processes 4.1, 4.2, 4.3 … on the level 2 diagram.

3. **Identify any data stores** that exist entirely within the level 2 boundary, and draw these data stores.

4. **Identify data-flows between the processes and data stores that are entirely within the level 2 system boundary**. Remember, every data store inside this boundary should have at least one input and one output date flow.

5. **Check the diagram**. Ensure that the level 2 data-flow diagram does not violate the rules for data-flow diagram constructs.

Making levels

For all systems it is useful to make at least two levels — the context diagram and the level one diagram. In fact, when in the earlier description of how to create DFDs you were told to start by identifying the external entities and then to identify the inputs and outputs of the system, you were learning how to produce the context diagram. The rest of the description was how to produce the level one diagram.

Whenever you perform data-flow modeling, start in exactly this way, producing a context diagram and then a level one diagram. Of course, in producing the level one diagram you may realise you need more inputs and outputs and possibly even more external entities. In this case, simply add the new data-flows and the new entities to the level one diagram and then go back and add them to the context diagram so that both diagrams still balance. Conversely, you may realise that some of the inputs and outputs you originally identified are not relevant to the system. Remove them from the level one diagram and then go back to the context diagram and make it balance by removing the same inputs and outputs.

This constant balancing between diagrams is very common when doing levelling.

What about making more levels? There are two reasons for making more levels. The first is the obvious one: you, as the software engineer, have not fully described a process to your satisfaction, so you expand that process into a next level diagram. The new diagram is built in just the same way that a level one is built from a context diagram only the new inputs and outputs are precisely to the data flows to and from the process you are expanding.

The second reason is that you realise the diagram you are working on is becoming cluttered and unclear. To simplify the diagram, collect together a few of the processes. Ideally, these processes should be related in some way. Replace them with a single process and treat the original collection of processes as a lower level, expanding the new process. The inputs and outputs to the new process are whatever inputs and outputs that are needed to make the diagrams balance. Remember to re-number the old processes to show that they have been moved down a level.

When doing this, if there is a data-store which interacts with these processes, and only these processes, then this too can be put on the lower level diagram.

Do not group random processes together to make a lower level diagram. This will only end up in a tangle of arrows and unrelated processes. A good guide as to whether or not you have chosen a sensible collection is try coming up with a new name for the replacement process. If you cannot do this then you have probably made too general a grouping. Perhaps leave out one or two processes or try a different grouping.

Always bear in mind that levelling is meant to simplify and clarify the diagrams, and if this cannot be done then it may be best to leave the diagram as it is.

Balancing

The key to successfully levelling is to make the diagrams balance. For example, if a second level diagram expands a first level process then all the inputs to the process must be inputs to the second level diagram, and all the outputs from the process must be outputs on the second level diagram. Moreover, there must be no other inputs and outputs. To be particular, all the inputs and outputs of the system which appear on the context diagram must appear on the level one diagram and there should be no other inputs and outputs on the level one diagram.

This does not mean there can be no changes to the higher levels of a set of diagrams. When producing a lower level diagram, the software engineer may realise that a new input is needed for the process to be able to carry out its task. In which case, the software engineer should add this data-flow as an input and then add the input as a data-flow to the original process. If needs be, this input may be added at several levels higher up. The software engineer may add new outputs in the same way.

As long the diagrams always balance, inputs and outputs can be added and removed wherever necessary.

Numbering

Numbering in a levelled set of diagrams is important, as the numbers help you to find your way around the levels. It is easily described by example. Suppose *Receive Order* is the process numbered 3 on the level one diagram (remember, numbers do not indicate any order, they are simply labels) and this is expanded to a level two diagram. The process numbers on the level two diagram will be 3.1, 3.2,

3.3 and so on. Suppose now that process 3.4 on the level two diagram is *Register New Customer* and needs further expansion to a level three diagram. The process numbers on this diagram will be 3.4.1, 3.4.2, 3.4.3 and so on. The rule used here is this: *if X is the number of the process you wish to expand, then the numbers on the next level are X.1, X.2, X.3...*

The same applies for data stores. Data stores that appear in a level two diagram expanding a process labelled 4 in the level one diagram will be numbered D4.1. D4.2, D4.3 and so on. Deeper levels will be D4.1.1, D4.1.2, the numbering scheme being just the same as for processes.

Note though, it is not the data stores that are expanded. They may simply appear in the expansion of a process.

Process descriptions

A software engineer may define a process where no further expansion is appropriate because there are no separate sub-processes which may make up the original process. However, the software engineer may still wish to describe the process in more detail as it is a particularly difficult or tricky process. In this case, the software engineer writes down a process description for the process. This can take any form which the software engineer thinks appropriate. Traditional flowcharts could be used or plain English. More common is what is called structured English. This looks like English only it is written more like a computer language. It used to avoid the problem that different people reading the same piece of plain English may understand it in different ways.

Validation

It should be clear that producing data-flow diagrams can be complicated. A routine check using the following questions should make sure that you find any simple mistakes. The first set of questions refer to a single diagram, so if you have a set of levelled data-flow diagrams then these checks need to be made for each diagram.

1. Is every data-flow attached to a process at either the beginning or the end of the arrow?

2. Is every data-flow labelled with a sensible noun?

3. Does every process have at least one input and at least one output?

4. Is every process named sensibly (no uses of words such as "process" or "handle") with an action and what is acted upon? (The template is "Do something to something")

5. Is every data store named with the type of thing it stores in the plural?

6. Where data stores and external entities have been shown several times on one diagram, do all instances have a "diagonal" line?

7. Are there any data-flows which cross? If so, try and add more duplicate external entities or data stores to avoid the crossing.

This second set of questions is specifically about levelling and so should be asked about the set of diagrams as a whole.

1. Do all diagrams balance? That is, where a diagram expands a process in a higher level, are the inputs and outputs to the process identical to the inputs and outputs on the expanded, lower level diagram?

2. Are all external entities shown on both the context diagram and level one diagram?

3. Are all of the processes and data stores numbered correctly?

All data-flow diagrams are an aid to communication between software engineers and their customers. Although they may be correct and accurate, a messy or tangled data-flow model will reduce communication as surely as a long-winded text description. To avoid this, as the diagrams evolve, re-

draw them whenever they begin to get cluttered or have several corrections on them. A simple re-arrangement of the components may be sufficient to greatly improve a diagram.

An example in constructing a data-flow diagram

Just as for logical data structures, to make a data-flow diagram we must analyse the requirements and describe the system in terms of the components of data-flow diagrams. Several attempts will be needed before a final and complete model of the system can be produced.

Unfortunately, there is no straightforward way to progress through developing a data-flow diagram. We are aiming therefore to build a skeleton model on paper which we can then work with and develop more fully. Each stage will be illustrated with examples from the following Estate Agent case study (repeated from Chapter 4, *An Introduction to Analysis and Design*).

Estate Agency case study

Clients wishing to put their property on the market visit the estate agent, who will take details of their house, flat or bungalow and enter them on a card which is filed according to the area, price range and type of property.

Potential buyers complete a similar type of card which is filed by buyer name in an A4 binder.

Weekly, the estate agent matches the potential buyer's requirements with the available properties and sends them the details of selected properties.

When a sale is completed, the buyer confirms that the contracts have been exchanged, client details are removed from the property file, and an invoice is sent to the client. The client receives the top copy of a three part set, with the other two copies being filed.

On receipt of the payment the invoice copies are stamped and archived. Invoices are checked on a monthly basis and for those accounts not settled within two months a reminder (the third copy of the invoice) is sent to the client.

Identify the system boundaries

The easiest place to making a data-flow model of a system is to identify what the external entities of the system are and what inputs and outputs they provide. These give you the boundary between the system and the rest of the world.

External entities must provide inputs or receive outputs. There are usually one or two which stand out as obviously interacting with the system but not being part of the system. In the Estate Agent system, *Client* and *Buyer* stand out as good candidates for external entities. Others may be harder to spot, but once again consider nouns in the case study and add them to a list of possible external entities.

It may be tempting to add *Estate Agent* as an external entity as it obviously interacts with the system. However, the estate agent is in fact part of the system in that he or she manipulates the data within the system. Another way to think about it is that the estate agent will actually be replaced by the new software system and so does not need to appear in the data-flow diagram.

From the list of candidates of external entities, determine what inputs they provide and what outputs they receive. If a candidate entity does not seem to provide data into the system or receive data from the system then it is not an external entity and can be discounted (for now).

An external entity stands for the type of thing interacting with the system so all clients and all buyers are represented by the *Client* and *Buyer* external entities.

Having identified the external entities there are two ways of progressing from here. Both are equally sensible approaches and are covered in the next two subsections.

Follow inputs

Each input to the system must be received by a process. This gives us a natural way to start building up the model.

First, take one of the more significant external entities and one of the main inputs it provides. In our case, a *Client* providing *Property Details* is a good place to start. Draw an oval for the entity, a data-flow for the input and a process which receives the input. From the case study there should something that suggest what happens when this data comes in and this will be the name of the process. For *Property Details*, the case study says that the estate agent enters the details on a card and files them. So the process name should be either *Record Details* or *Receive Details*.

Every process must have at least one output, so, for the process in hand, consider what the outputs must be and put labelled arrows on the diagram for the outputs. The data must be changed by a process and so should have a different name from the input data. *Property Details* are taken and recorded as a property on the file so the output could be just something like *Recorded Property* or more simply, *Property*.

Now start again only using this output as a new input. It must either go to another process, to a data store or to an external entity. It should be clear from the case study what happens.

If a new process is needed then do the same again. Find a sensible name for the process using the case study, determine and label the outputs and then follow the outputs.

If the data is stored then add a data store to the diagram, name it sensibly from the case study and draw the output arrow going into the data store. This is what happens in *Property* and so we add the data store *Properties*.

If the output is an output from the system then simply add the external entity which receives the output.

When the data is finally output or comes to rest in a data store, go back and follow any of the other outputs which may have been defined on the way. When they are exhausted, choose a new input and follow that through in exactly the same way.

Follow events

Another way to approach building up a data-flow model is to consider what happens in the system. The case study will outline a number of events. There must be processes in the system which respond to these events or even make them happen. Identify these processes and then add the data inputs which are used by the process and determine the outputs.

For example, in the estate agent example, there is the phrase, "When a sale is completed...". This is an event: a sale is completed. From the case study we see that lots of things then happen: the buyer confirms exchange of contracts so this is an input to some process; the client details are removed from the file and invoice is sent out. This is the process. A sensible name might be *Record Sale* or possibly *Receive Sale Confirmation*. The data needed is the input from the *Buyer* and *Client* details which are on file. This must mean there is a data store somewhere on the diagram holding this information. If there is not one there already then add it. And the output must be an invoice to the *Client*.

From here on the approach is the same as following inputs. For any new outputs, work out where those outputs must go and if it is to a process follow them as if they were inputs to the new process.

Most processes can be found in the case study using either technique of following inputs or following events. However, some processes are related to temporal events and so can only be found by following events.

As the name suggests, temporal events are events which occur at specific times. They are not prompted to happen by the arrival of new data, but rather because a certain time has been reached. These events often appear in case studies beginning with phrases such as, "Once a month..." or, "At the end of every day,...". However, once these have been identified, producing the model by following this event is exactly the same as for any other event.

In the estate agent system, there are two temporal events: there is a weekly matching of potential buyers with properties; invoices and reminders are sent out on a monthly basis.

Though time is the trigger the processes carrying out temporal events, time is generally not shown on the data-flow diagram. This is because the time aspect is often just a practical implementation rather than rigid necessity. For example, the matching of buyers and properties at the estate agents need not be weekly. It is probably done weekly so that it always gets done, and also so that it does not interrupt the other daily business. With an automated system it may be possible to match buyers with properties as soon as any new details on either arrive.

Where time is crucial to a process, say accounting done at the end of a financial year, then this can be reflected in the name of the process. For example, "Calculate end of year profits".

Fill in gaps

After building a model that handles each input or each event, it is worth going over the processes defined so far.

For each process, ask the question, "Does this process have all the information it needs to perform its task?" For instance, if a process sends out invoices, does it have all the details of the invoice and the address of where the invoice should go? If the answer is *No*, then add a data-flow into the process which consists of the data needed by the process. If there are several, clearly distinct items of data needed, then you may need an arrow for each item. Now try to identify the source of the data.

First, see if the data can be found already inside the system, either on a data store or as a result of a process. If not, it may be that the data can be obtained by processing some of the existing data in, which case add a new process that takes the existing data and makes the data you require. Or, the data may be available but from the case study it is clear that there is a time-lag between the process that produces the data and the process which uses it. Simply add a data store where the data can reside till it is needed.

If there is still no source for the data then it could be from an external entity. In which case, this is a new input to the system. It may not be explicitly mentioned in the case study, but if it is necessary then it should be added. Having added the new input from the appropriate entity, go back and correct the context diagram.

This is an important task. If there is not enough data to support a task then the system will not function properly. Of course, to be on the safe side, you could have all the data going to all the processes! But this is not really a solution because with a large system this would be impractical.

Having examined all the processes, check that all the outputs have been generated. All of the inputs should have been covered already, but this does not mean that all the outputs have been produced. If there is still an output which does not appear on the diagram, see if there is a process where it could come from. If there is no sensible candidate, add a process and begin to work backwards. What inputs does the new process need? Where do these inputs come from? This task is almost the same as the one just described.

Any left over outputs must have come from a process. Outputs cannot come from data stores or external entities. If there is no sensible way to fit the output into the diagram then it may be that it is not a sensible output for the system you are currently considering. Use the case study to confirm this.

Finally, check the data stores. Data must enter a data store somehow and generally data on a data store is read. For each data store, identify when the store is either written or read by considering the processes which may use the data. Also, use the case study to see that you have not missed any arrows to or from a data store.

Repeat

By this stage, you will have considered all the inputs, all the outputs and produced a first draft of the data-flow model of the system.

Review the case study, looking for functionality described which is not performed by the model. In particular, look for temporal events as these are sometimes hidden implicitly in the text.

Where necessary, add new processes that perform the omitted functions and use the method of following events to work out their inputs and outputs. Fill in the gaps of the model in exactly the same way as was done to produce the first attempt.

The model can be declared finished when you have considered every word in the case study and decided that it is not relevant or that it is incorporated in some way into the model.

Review

Questions

Review Question 1

Describe the two main ways in which data-flow diagrams are used to manage the complexity of systems.

A discussion of this question can be found at the end of this chapter.

Review Question 2

What are the four different system models which may include data-flow diagrams?

A discussion of this question can be found at the end of this chapter.

Review Question 3

What are the external entities in the following diagram Video-Rental LTD case study.

Figure 6.11. Find the external entities
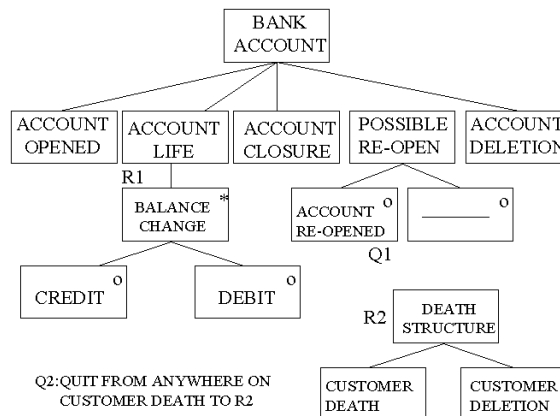
A discussion of this question can be found at the end of this chapter.

Review Question 4

What are the data-flows between Supplier and Video-Rental LTD case study in the above diagram?

A discussion of this question can be found at the end of this chapter.

Review Question 5

What are the processes in the above diagram Video-Rental LTD case study?

A discussion of this question can be found at the end of this chapter.

Review Question 6

What are the data stores in the context diagram Video-Rental LTD case study?

A discussion of this question can be found at the end of this chapter.

Review Question 7

What does the zero mean in the top left of the Video-Rental LTD process in the context diagram?

A discussion of this question can be found at the end of this chapter.

Review Question 8

Describe the first, top level DFD created for a system.

A discussion of this question can be found at the end of this chapter.

Review Question 9

Outline the main roles of Context Diagrams.

A discussion of this question can be found at the end of this chapter.

Review Question 10

Follow the suggested steps to create a context diagram for the Video Rental LTD case study.

A discussion of this question can be found at the end of this chapter.

Review Question 11

The following Estate Agency case study will be used in this, and some later, review questions. This is the same case study as used in Chapter 4, *An Introduction to Analysis and Design*, but we will repeat the text here for your convenience.

Estate Agency case study

Clients wishing to put their property on the market visit the estate agent, who will take details of their house, flat or bungalow and enter them on a card which is filed according to the area, price range and type of property.

Potential buyers complete a similar type of card which is filed by buyer name in an A4 binder.

Weekly, the estate agent matches the potential buyer's requirements with the available properties and sends them the details of selected properties.

When a sale is completed, the buyer confirms that the contracts have been exchanged, client details are removed from the property file, and an invoice is sent to the client. The client receives the top copy of a three part set, with the other two copies being filed.

On receipt of the payment the invoice copies are stamped and archived. Invoices are checked on a monthly basis and for those accounts not settled within two months a reminder (the third copy of the invoice) is sent to the client.

Create a context diagram for this Estate Agency case study.

A discussion of this question can be found at the end of this chapter.

Review Question 12

What are the processes in the level 1 DFD for the Video Rental case study below?
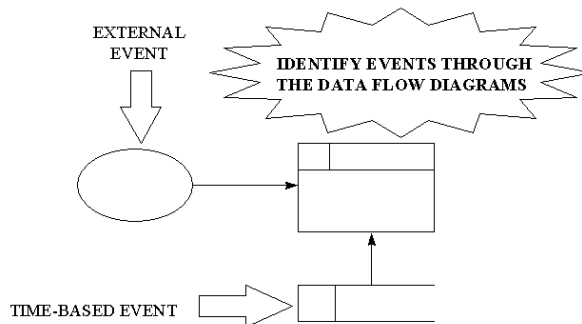
A discussion of this question can be found at the end of this chapter.

Review Question 13

What are the data stores in the level 1 DFD above?

A discussion of this question can be found at the end of this chapter.

Review Question 14

What is meant by functional decomposition?

Under what conditions would you decompose a process on a Data-Flow Diagram?

A discussion of this question can be found at the end of this chapter.

Review Question 15

Decompose the Video Rental Level 1 DFD process "loan of video" into a Level 2 DFD.

A discussion of this question can be found at the end of this chapter.

Review Question 16

Create a Level 1 DFD for the Estate Agency case study based on the context diagram from the previous Review Question and the case study text.

A discussion of this question can be found at the end of this chapter.

Review Question 17

Create a Level 2 DFD for the "invoice client" process of the Estate Agency case study based on the Level 1 DFD from the previous Review Question and the case study text.

A discussion of this question can be found at the end of this chapter.

Review Question 18

What are some of the specific benefits of Data Flow Models?

A discussion of this question can be found at the end of this chapter.

Review Question 19

Describe each of the main elements of Data-Flow Diagrams.

A discussion of this question can be found at the end of this chapter.

Review Question 20

Describe two of the points at which Data-Flow Diagrams are used during systems analysis

A discussion of this question can be found at the end of this chapter.

Review Question 21

The details of any level 2 or lower DFD could be displayed in a level 1 DFD, so really there is no reason not to model the entire system in a single level 1 DFD and avoid all the problems of balancing and hierarchical process numbering and so on.

A discussion of this question can be found at the end of this chapter.

Review Question 22

There is no facility in the Data-Flow Modelling technique to model the order in which processes occur and data flows. When creating an information system such time-based aspects of a system are just as important as the processes and data themselves.

Why do you think that such a feature not been created as part of Data-Flow Diagrams, and how can system designers get around this omission?

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Decomposition – which divides complex information into manageable chunks using a hierarchical tree structure. An overview of the problem is presented at the top level of the structure, while lower levels provide increasing depth of detail for narrower areas of the problem

Abstraction – enables software engineers to concentrate on only one aspect of the system at a time. Different models are used to model different perspective of the system. Data-Flow Diagrams concentrate on information flows and the activities which process this information.

Discussion of Review Question 2

Current System Physical model – the physical processes and data-flows and data stores of the current system may be modelled with DFDs (e.g. forms, pieces of paper, physical files and filing systems etc.)

Current System Logical model – the logical processes and data-flows and data stores of the current system may be modelled with DFDs (e.g. logical actions, logical collections of data, logical packages of information flowing etc.)

Required System Logical model – the logical processes and data-flows and data stores of the required system may be modelled with DFDs as part of the specification of the required system

Required System Physical model – the physical processes and data-flows and data stores of the required system may be modelled with DFDs as part of the design for the required system

Discussion of Review Question 3

There are two external entities shown in the above diagram (as ovals):

• Customer – a customer who can borrow videos

• Supplier – the local supplier

Discussion of Review Question 4

There are 3 data-flows shown in the above diagram (as named arrows):

• Available titles – from Supplier to Video-Rental LTD

• Order – from Video-Rental LTD to Supplier

• Videos – from Supplier to Video-Rental LTD

• Supplier – the local supplier

Discussion of Review Question 5

There is just one process in the above diagram (a rectangle with three parts) - Video-Rental LTD

Discussion of Review Question 6

There are no data stores in the above diagram (rectangles with two parts)

Discussion of Review Question 7

The top left part of a process rectangle is the process number. For context diagrams, if any number at all is used, it is usually zero. The zero indicates that this is the whole system, whereas in lower level DFDs numbers like 1 and 3 indicate sub-processes of the whole system. This will become more clear when you have progressed to understanding and creating hierarchical, levelled diagrams.

Discussion of Review Question 8

A Context diagram is the first DFD to be created for a system. It represents a model of the system as a whole (i.e. as a single process) and this systems interactions with external entities that are outside the boundaries of the system, but which provide inputs to, and receive the outputs of the system being modeled.

Context diagrams have the following features:

• only one process, representing the whole system

• they show no data stores

• they show all external entities with which the system exchanges data-flows.

Discussion of Review Question 9

Functional decomposition is the breaking down of higher level processes into their component sub-processes, data-flows and data stores as lower level DFDs.

The condition to decide to decompose a process is any time where there is some detailed aspect of the system that is not modeled by the process description alone — i.e. when a lower level DFD provides something more to the software engineer, such as sub processes, additional data stores, and data-flows that are used only for the process and which have not been modeled at the higher level DFD.

Discussion of Review Question 10

Identify data-flows by listing the major documents and information flows associated with the system.

You may find the use of the following kind of table is useful:

data-flow	Sender	Receiver

From the case study we can underline all potential data flows INTO AND OUT OF THE SYSTEM. At this point look for any possible data-flows, we can change our minds at any time in the process of creating a context diagram. We are not worried about data-flows that seem to be within the system at present, so the sender and receiver should always be either an external entity, or the system itself.

Video-Rental LTD is a small video rental store. The store lends videos to customers for a fee, and purchases its videos from a local supplier.

A customer wishing to borrow a video provides the empty box of the video they desire, their membership card, and payment – payment is always with the credit card used to open the customer account. The customer then returns the video to the store after watching it.

If a loaned video is overdue by a day the customer's credit card is charged, and a reminder letter is sent to them. Each day after that a further chard is made, and each week a reminder letter is sent. This continues until either the customer returns the video, or the charges are equal to the cost of replacing the video.

New customers fill out a form with their personal details and credit card details, and the counter staff give the new customer a membership card. Each new customer's form is added to the customer file.

The local video supplier sends a list of available titles to Video-Rental LTD, who decide whether to send them an order and payment. If an order is sent then the supplier sends the requested videos to the store. For each new video a new stock form is completed and placed in the stock file.

data-flow	Sender	Receiver
video	system	customer
customer detail	customer	system
membership card	customer	system
membership card	system	customer
empty video box	customer	system
payment	customer	system
return of video	customer	system
credit card charge	system	customer (or credit card firm)
overdue reminder letter	system	customer
available titles	supplier	system
order	system	supplier
payment	system	supplier
requested videos	supplier	system
stock form	system	system

Let us consider each data-flow in turn:

- **video by customer when joining the store** — this is a strong candidate data-flow, though we might name it 'video loan' or 'details of loaned video'

- **customer details by customer when joining the store** — this is a strong candidate data-flow

- **membership card issued to customer** — this is a strong candidate data flow

- **membership card presented by customer when renting a video** — this is a strong candidate data-flow

- **empty video box presented by customer when renting a video** — this is a strong candidate data-flow, but perhaps should be call 'request for video' or something similar

- **payment by customer when renting a video** — this is a strong candidate data flow

- **return of video by customer** — this is a strong candidate data flow, although the data might be 'returned video' or 'returned video details'

- **credit card charge by system** — this is a strong candidate data flow, but in fact we have already identified a payment by the customer (when renting a video) and we could just consider this to be anther example of customer payment (for simplicity, although alternatively we could consider this a separate data-flow, the decision could be influenced on the sophistication of the systems processing of payments, and might be delayed until more detailed DFDs are produced later in the analysis procedure)

- **overdue reminder letter from system** — this is a strong candidate data flow

- **payment by system for order** — this is a strong candidate data flow

- **list of available titles from supplier** — this is a strong candidate data flow

- **the requested videos from supplier** — this is a strong candidate data flow, although might be called something like 'videos purchased'

- **stock form** — this last data-flow is within the system, so this will not be used in the context diagram but will probably appear in a more detailed DFD later

You might have noticed

- **Identify external entities** by identifying sources and recipients of the data-flows, which lie outside of the system under investigation.

  This step is easy if we have created a table like the above, since we can just create a list of all the different entities:

  - customer

  - supplier (a candidate might be the credit card company, but we shall choose to consider the customer to be charged in this case for simplicity)

  Draw and label a process box representing the entire system.

- **Draw and label the external entities** around the outside of the process box.

  We just need to add external entity symbols for 'customer' and 'supplier'.

- **Add the data-flows between the external entities and the system box**

  we now need to add those data-flows earlier:

data-flow	Sender	Receiver
video loan	system	customer

data-flow	Sender	Receiver
customer details	customer	system
membership card	customer	system
membership card	system	customer
request for video	customer	system
payment	customer	system
return of video	customer	system
overdue reminder	system	customer
available titles	supplier	system
order	system	supplier
payment	system	supplier
requested video	supplier	system

We can do a quick check when we have created the diagram by counting the number of flows out of, and into each entity.

From column 'sender' we can see there should be:

- 5 data-flows out of the system

- 5 data-flows out of customer

- 2 data-flows out of supplier

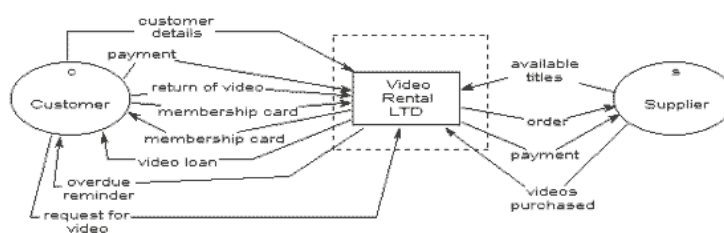From column 'receiver' we can see there should be:

- 7 data-flows into the system.

- 3 data-flows into customer

- 2 data-flows out of supplier

Our context diagram looks as follows:

Discussion of Review Question 11

There are 3 data-flows shown in the above diagram (as named arrows):

- Create new customer
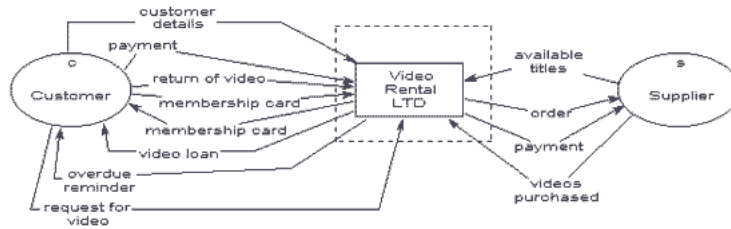
- Loan of video

- Stock control

Discussion of Review Question 12
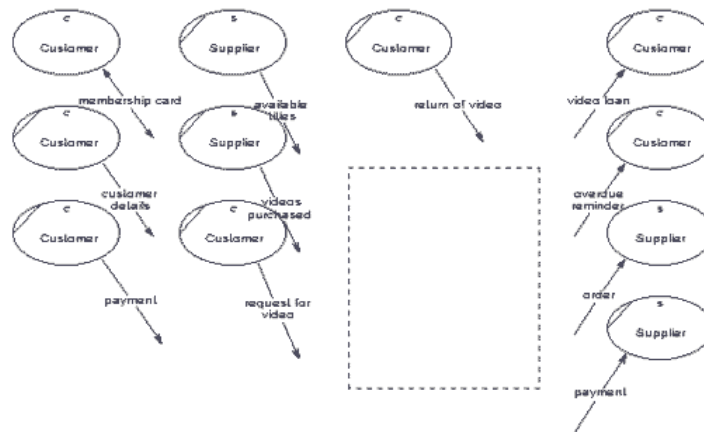
There are 2 data stores:

- Stock file

- Stock file

Discussion of Review Question 13

First we start with the context diagram, since all external entities and data-flows on this diagram must appear on our Level 1 DFD:

We can now create an 'empty' Level 1 DFD with these entities and data-flows:

- **Identify processes**. Each data-flow into the system must be received by a process. Each process must have at least one output data-flow. Each output data-flow of the system must have been sent by a process.

Now we need to identify the recipient and sending processes of the system for each data-flow. We need to replace with a system process each occurrence of 'system' as the sender or recipient in the table of data-flows created previously.

Possible processes have been inserted in the following table:

Data-Flow	Sender	Customer
video loan	system - loan of video	customer
customer details	customer	system - create new customer
membership card	customer	system - loan of video
membership card	system - create new customer	customer
request for video	customer	system - loan of video
payment	customer	system - loan of video
return of video	customer	system - loan of video
overdue reminder	system - loan of video	customer

Data-Flow	Sender	Customer
available titles	supplier	system - stock control
order	system - stock control	supplier
payment	system - stock control	supplier
requested videos	supplier	system - stock control

- **Draw the data-flows between the external entities and processes**. After creating process boxes and drawing the data-flows the diagram looks as follows:
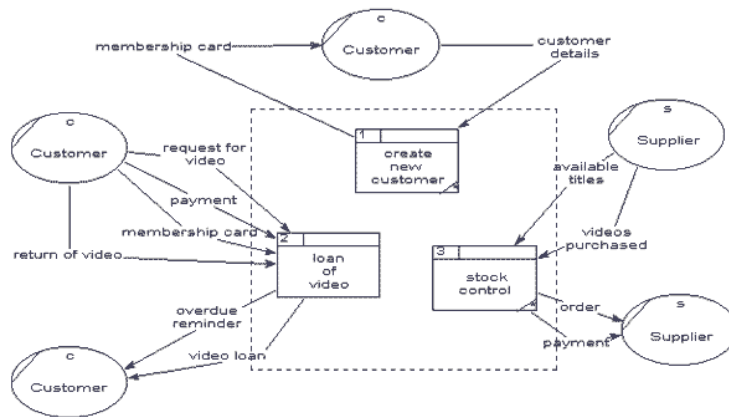
- **Identify data stores** by establishing where documents / data needs to be held within the system. Add the data stores to the diagram, labelling them with their local name or description.

  There seem to be 2 main data stores required: a store of customer details 'customer file' and a store of which videos are in stock 'stock file'.

  After adding these to the diagram looks as follows:
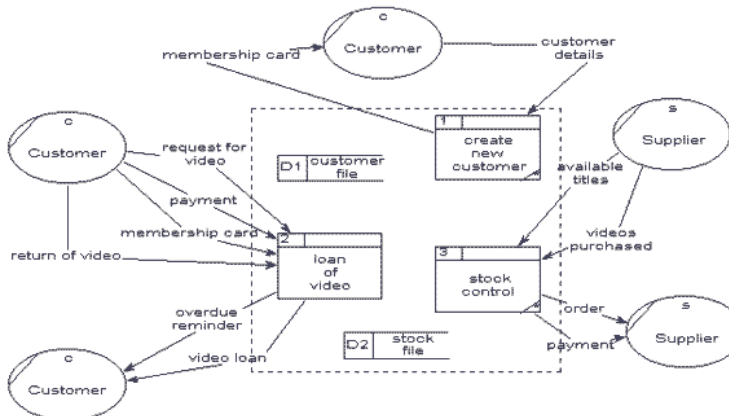
- **Add data-flows flowing between processes and data stores** within the system. Each data store must have at least one input data-flow and one output data-flow.

  We can create a table to indicate which processes send and receive data from each data store:

Data store	data-flow IN FROM	data-flow OUT TO
customer file	customer details FROM create new customer	customer details TO loan of video
stock file	new video details FROM stock control	overdue items TO loan of video

After adding these data-flows the diagram looks as follows:
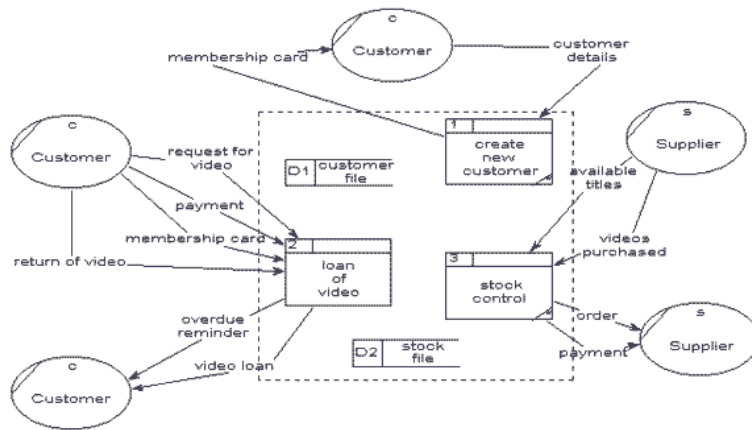
- **check diagram** No record seems to be made of when a video is lent to a customer — there ought to be a data-flow from 'loan of video' to 'stock file' called something like 'item on loan'. Likewise when an item is returned the details should be recorded in a data-flow called something like 'item returned'.

Apart from these extra two data-flows the diagram appears to be correct.

So our Level 1 DFD for the Video Rental case study is now:
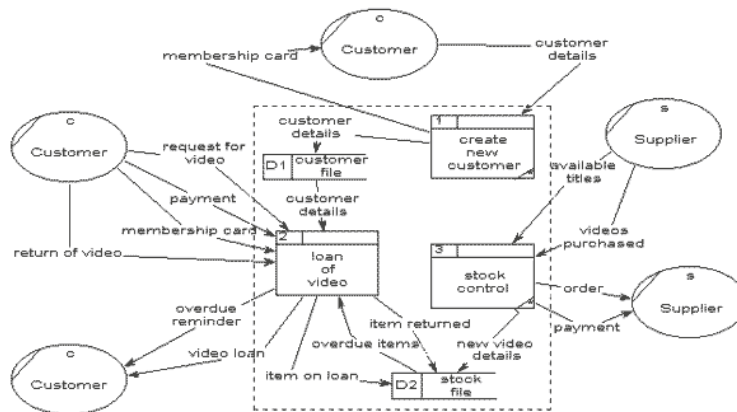
Discussion of Review Question 14

Make the process box on the Level 1 diagram the system boundary on the Level 2 diagram that decomposes it.

This gives us the following, "empty" Level 2 DFD:
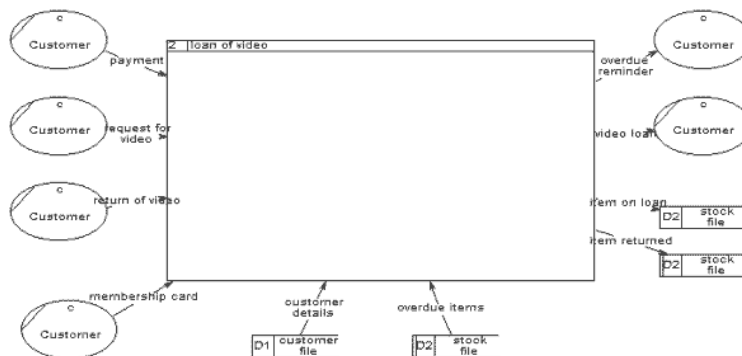
Identify the processes inside the Level 2 system boundary and draw these processes and their data-flows.

For each data-flow into and out of the process for which this Level 2 diagram is being created we need to identify an appropriate sub-process to receive and send the data flows. The following table lists each data-flow and suggests a suitable sub-process to receive/send the data-flow:

| data-flow	Sender	Receiver
video loan	loan of video - process loan	customer
membership card	customer	loan of video - validate customer
request for video	customer	loan of video - validate customer
payment	customer	loan of video - issue video
return of video	customer	loan of video - restock video
customer details	customer-file	loan of video - validate customer
overdue items	stock-file	loan of video - process late return
item returned	loan of video - restock video	stock-file
item on loan	loan of video - issue video	stock-file
overdue reminder	loan of video - process late return	customer

Adding these processes and data-flows to the diagram we get the following:
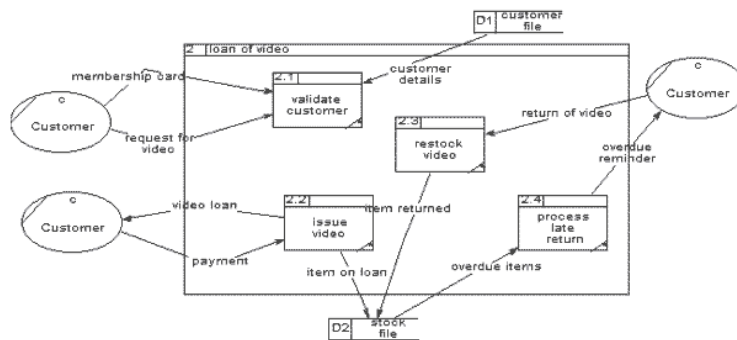
Identify any data stores that exist entirely within the Level 2 boundary, and draw these data stores: For this example there don't appear to be any "local" data stores

Identify data-flows between the processes and data stores that are entirely within the Level 2 system boundary: Since there are no local data stores, there are no data-flows between processes and data stores to be added.

Check the diagram: Upon checking the diagram, we find that the process "validate customer" has no output data flows. Looking more closely we see that a plausible data flow out of "validate customer" would be something like "loan permission".

Upon adding this new data-flow the diagram looks as follows:
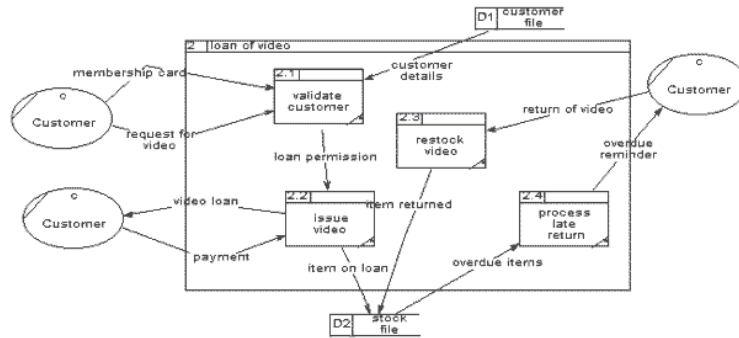
Discussion of Review Question 15

- **Identify data-flows** by listing the major documents and information flows associated with the system.

    You may find the use of the following kind of table is useful:

data-flow	Sender	Receiver

    From the case study we can underline all potential data flows INTO and OUT OF THE SYSTEM. At this point look for any possible data-flows, we can change our minds at any time in the process of creating a context diagram. We are not worried about data-flows that seem to be within the system at present, so the sender and receiver should always be either an external entity, or the system itself.

    Clients wishing to put their property on the market visit the estate agent, who will take details of their house, flat or bungalow and enter them on a card which is filed according to the area, price range and type of property .

    ## Note

    Potential buyers complete a similar type of card which is filed by buyer name in an A4 binder.

    Weekly, the estate agent matches the potential buyer' requirements with the available properties and sends them the details of selected properties.

    When a sale is completed, the buyer confirms that the contracts have been exchanged, client details are removed from the property file, and an invoice is sent to the client. The client receives the top copy of a three part set, with the other two copies being filed.

    On receipt of the payment the invoice copies are stamped and archived. Invoices are checked on a monthly basis and for those accounts not settled within two months a reminder (the third copy of the invoice) is sent to the client.

We can build a table of these data-flows, and the senders and receivers of these flows.

data-flow	Sender	Receiver
house details	client	system
buyer details	buyer	system
selected properties	system	buyer
contract	buyer	client
invoice	system	client

data-flow	Sender	Receiver
payment	client	system
reminder	system	client

Rejected candidates for data-flows include:

- the internal copies of the invoice - these data-flows do not go outside the system boundary so will not be part of this context diagram (but may feature on a more detailed DFD later)

- the client details card is filed IN the system, so this internal data-flow will not feature on the context diagram

It is worth noting that the exchange of contracts between client and buyer is not a data-flow into or out of the system, but this data-flow between external entities is relevant so ought to be notated on the context diagram.

- **Identify external entities** by identifying sources and recipients of the data-flows, which lie outside of the system under investigation.

  This step is easy if we have created a table like the above, since we can just create a list of all the different entities: client, buyer.

- **Draw and label a process box** representing the entire system:

- **Draw and label the external entities** around the outside of the process box. We just need to add external entity symbols for 'client' and 'buyer'

- **Add the data-flows** between the external entities and the system box. We now need to add those data-flows earlier. Our context diagram looks as follows:
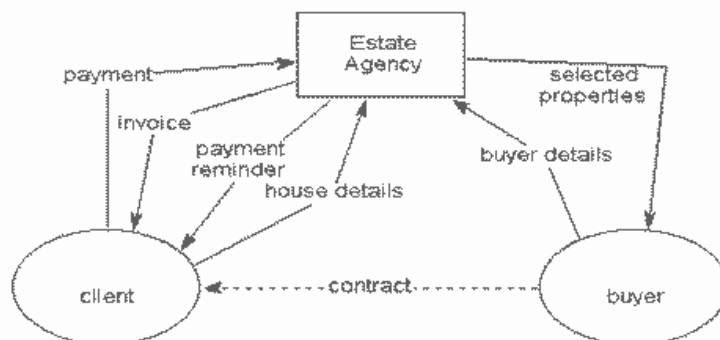
We can check the diagram quickly looking at the table:

data-flow	Sender	Receiver
house details	client	system
buyer details	buyer	system
selected properties	system	buyer
contract	buyer	client
invoice	system	client
payment	client	system
reminder	system	client

Client should send 2 data-flows, and receive 3.

Buyer should send 2 data-flows and receive 1.

System should send 3 data-flows and receive 3.

Discussion of Review Question 16

We should start with the context diagram, and create an 'empty' Level 1 DFD with all the same external entities and data-flows:
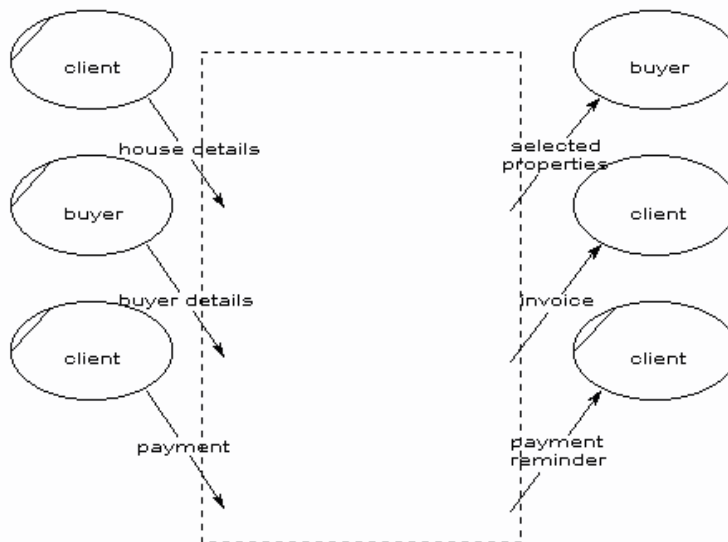
- **Identify processes** - Each data-flow into and out of the system must be received by /send by a process. Now you need to identify the recipient and sending processes of the system for each data-flow. We need to replace with a system process each occurrence of 'system' as the sender or recipient in the table of data-flows created previously. Possible processes have been inserted in the following table:

data-flow	Sender	Receiver
house detail	client	system - record new client
buyer details	buyer	system - record new buyer
selected properties	system - match properties	buyer
contract	buyer	client
invoice	system - invoice client	client
payment	client	system - archive sale
reminder	system - invoice client	client

- **Draw the data-flows between the external entities and processes.** We can now add these processes to the diagram, and connect the appropriate data-flows:
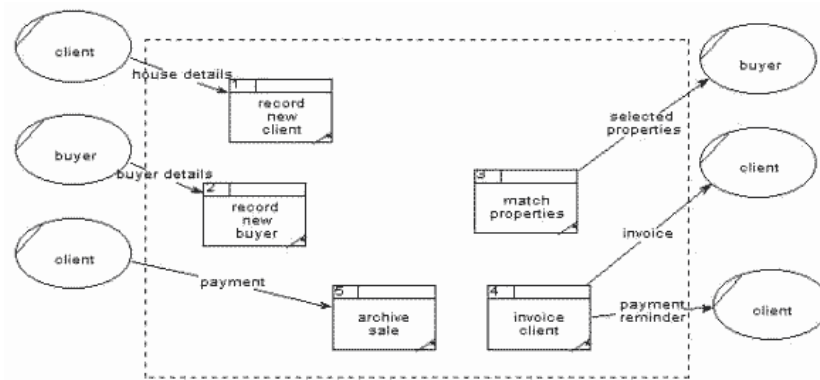
- **Identify data stores** by establishing where documents / data needs to be held within the system. Add the data stores to the diagram, labelling them with their local name or description. There are two 'card' stores (clients and buyers) so these should be data stores 'property file' and 'buyer details'. A file need to be kept for the invoice copies 'invoices'. We can add these data stores to the diagram:
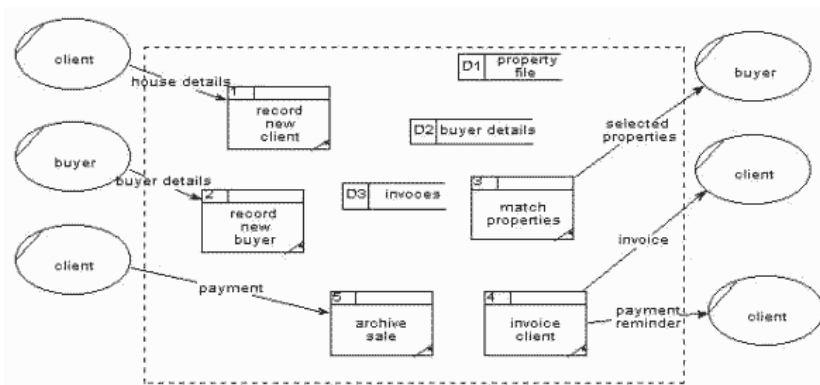
- **Add data-flows** flowing between processes and data stores within the system. Each data store must have at least one input data-flow and one output data-flow (otherwise data may be stored, and never used, or a store of data must have come from nowhere!). Ensure every data store has input and output data-flows to system processes. Most processes are normally associated with at least one data store.

We can create a table to indicate which processes send and receive data from each data store:

Data store	data-flow IN FROM	data-flow OUT TO
property file	property details FROM record new client	properties TO match properties
buyer details	buyer details FROM record new buyer	desired property TO match properties
invoices	invoice FROM invoice client	reminder TO invoice client

These data-flows can be added to the diagram:
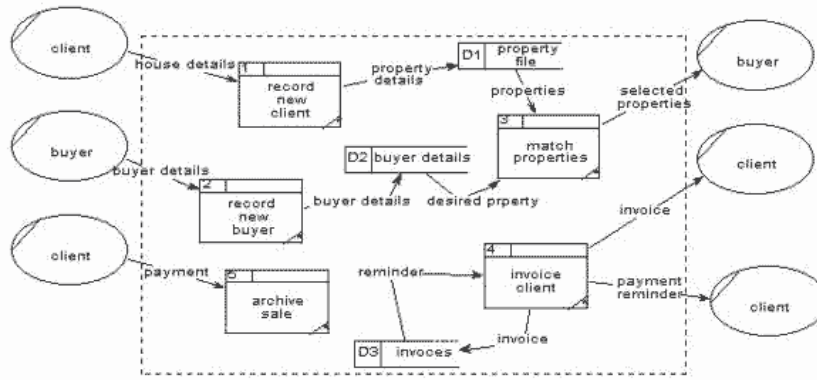
- **Check diagram**. We now can check the diagram for correctness, and find a process that has no output data-flow 'archive sale'. An appropriate data-flow, into data store 'invoices' would be something like 'record of payment'. The consistent and balanced Level 1 DFD now looks as follows:
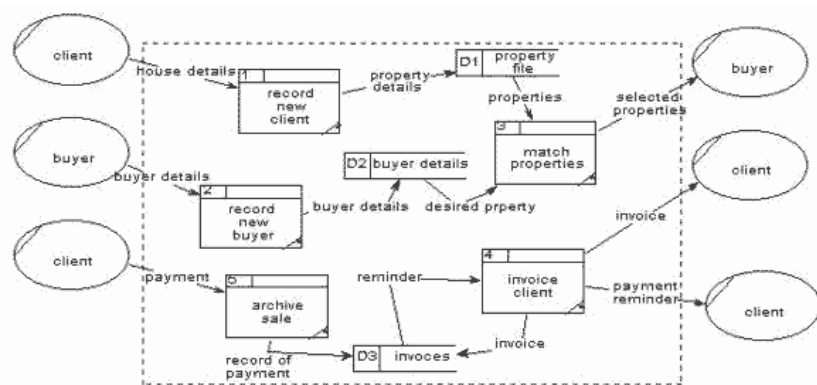
However, there is another problem with the diagram — what causes the process 'invoice client' to send an invoice or reminder to the client? The only input to the process 'invoice client' is a 'reminder' from the 'invoices' data store. The answer is that there are two things that trigger this process to send a data-flow to the client:

- knowledge that sale has been completed

- knowledge that a payment on an issued invoice is overdue

The second is a time-based event, and not modelled explicitly in Data-Flow Diagrams. However, the first indicates there should be a data-flow from an external entity to the system indicating that contracts have been exchanged. If we look carefully at the case study again, we find that:

Note

When a sale is completed, the buyer confirms that the contracts have been exchanged, client details are removed from the property file, and an invoice is sent to the client.

This must mean that the buyer informs the system that the sale is complete, so we must create a new data-flow from 'buyer' to 'invoice client' called something like 'confirmation of sale'. (NOTE: Since we are adding a new data-flow between the system and the external entities, we shall have to update the parent diagram — if we forget we will be reminded by any CASE tool consistency checker).

We also notice there should be a data-flow of 'client to delete' from process 'invoice client' to the data store 'property file'.

Our Level 1 DFD now looks as follows:
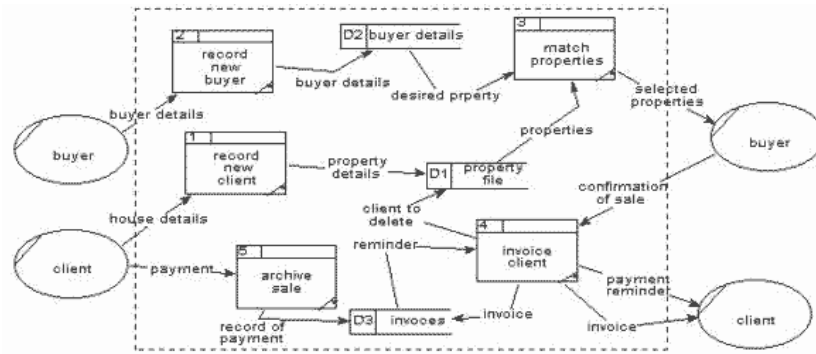
Discussion of Review Question 17

- **Make the process box on the Level 1 diagram the system boundary on the Level 2 diagram that decomposes it.**

  We should start with the Level 1 DFD, and create an 'empty' Level 2 DFD with all the same external entities and data-flows as the "invoice client" process.

  This gives us the following, "empty" Level 2 DFD:
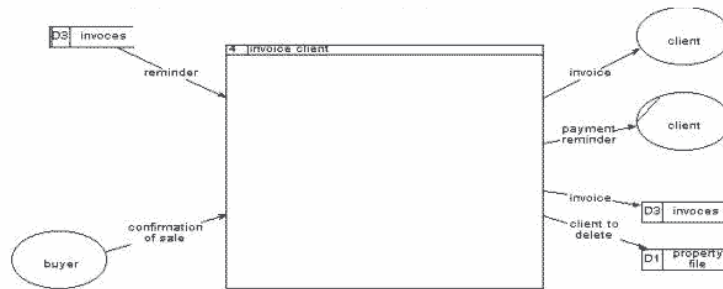
- **Identify the processes** inside the Level 2 system boundary and draw these processes and their data-flows.

  For each data-flow into and out of the process for which this Level 2 diagram is being created we need to identify an appropriate sub-process to receive and send the data-flows. The following table lists each data-flow and suggests a suitable sub-process to receive/send the data-flow:

data-flow	sender	receiver
invoice	invoice client - raise invoice	client
payment reminder	invoice client - process late payment	client
reminder	invoice client - process late payment	invoice client - process late payment
invoice	invoice client - raise invoice	invoices
confirmation of sale	buyer	invoice client - raise invoice
client to delete	*invoice client - ????*	*property file*

The last row in the table above is interesting — there doesn't appear to be a sub-process inside the "invoice client" process that creates the data-flow "client to delete". Looking carefully at the Level 1 DFD we can see that the "archive sale" process is probably most appropriate to be sending the property file the details of which client to delete, since it is this process that receives the payment

from the client. Therefore we need to delete this "client to delete" data-flow from the Level 2 DFD, and change the Level 1 DFD to have this data-flow from "achieve sale" to the "property file".

Adding these processes and data-flows to the diagram we get the following:
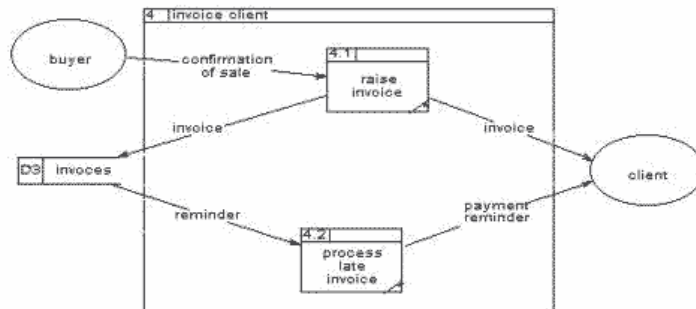
- **Identify any data stores** that exist entirely within the Level 2 boundary, and draw these data stores. For this example there don't appear to be any "local" data stores

- **Identify data-flows** between the processes and data stores that are entirely within the Level 2 system boundary. Since there are no local data stores, there are no data-flows between processes and data stores to be added.

- **Check the diagram.** There appear to be no inconsistencies with the diagram, so our final diagram stays the same.

Discussion of Review Question 18

Data-Flow Diagrams concentrate on the flow and transformation of data. High level Data-Flow Diagrams are decomposed to a set of more detailed diagrams.

Discussion of Review Question 19

- Processes – the activities carried out by the system.

- the data inputs and outputs to/from these activities.

- where information is stored within the system.

- the sources of data-flows into the system, and recipients of information leaving the system.

Discussion of Review Question 20

Any two of the following would be fine:

- Current System Physical model – the physical processes and data-flows and data stores of the current system may be modeled with DFDs (e.g. forms, pieces of paper, physical files and filing systems etc.) [investigating current system]

- Current System Logical model – the logical processes and data-flows and data stores of the current system may be modeled with DFDs (e.g. logical actions, logical collections of data, logical packages of information flowing etc.) [investigating current system]

- Required System Logical model – the logical processes and data-flows and data stores of the required system may be modeled with DFDs as part of the specification of the required system

- Required System Physical model – the physical processes and data-flows and data stores of the required system may be modeled with DFDs as part of the design for the required system

Discussion of Review Question 21

While, theoretically, it would be valid to model an entire system in a single level 1 DFD, for any non-trivial system such a diagram would fill an entire wall or floor of a very large room !

In the same way the a physical organisation is divided into departments or sections (or faculties for a University), to gain the benefits of local organisation and ability to manage, so levelled DFDs allow different logical functions of organisations to be abstracted and modelled together. So all sales functions of an organisation can be modelled as a single "sales" process, and then described at a lower level in more detail.

Obviously a major advantage of levelling is that the complexity of any single diagram can be restricted so as not to overwhelm the reader.

Discussion of Review Question 22

It should be remembered that each modelling technique (such as Data-Flow Diagrams and Entity-Relationship Diagrams) only presents one aspect of the system — the model of the complete system is formed when a number of different models are put together.

The three main traditional modelling techniques for systems analysis and specification are:

• Data-Flow diagrams

• Entity Relationship diagrams

• Entity Life histories

The third of these, Entity Life Histories (ELHs), is the modelling technique that represents those aspects of a system that change over time. Entity Life Histories are introduced in a later unit, and play an important role in relating the processes and data stores of DFDs with the logical data models of Entity Relationship Diagrams.

Any particular modelling technique will have been designed to represent only certain aspects of a system, since any non-trivial system would be much to complex to ever model with a single technique — any resulting diagram or set of diagrams would contain more information that would be usefully understandable by users and system developers.

Chapter 7. Design

Objectives

At the end of this chapter you will have acquired practical and theoretical knowledge and skills about modern software design. After successfully completing this module you should be able to:

• Describe the importance of abstraction and information hiding.

• Describe how abstraction and information hiding are used to handle changes in the software, and in testing the software.

• Show how information hiding and abstraction relates to the software's architecture.

• Define what a design pattern is.

• Give examples of various design classes.

Introduction

The last few chapters have introduced the concepts of analysis and design, along with various modelling techniques appropriate to these activities. We have also discussed some general topics, such as ensuring that your models are readable, that they are produced iteratively and incrementally to better accommodate change both in their requirements and in your own understanding of these requirements. In this chapter we will be discussing some general design guidelines which you may want to keep in mind when designing software systems. We will break these down into general areas such as *abstraction* and *architecture*, each dealt with in their own section.

Abstraction

Abstraction is the activity of reducing the information in a problem to only that information which is important to us. When we perform analysis and design we use many levels of abstraction in order to make both the problem we are solving and the software we are developing understandable and meaningful. At *higher abstraction levels*, the problem and the software are described in less detail and more broadly. At *lower abstraction levels*, more details are given, and the software begins to take on a more concrete form.

Ultimately, the software itself is the lowest level of abstraction that a software engineer will use. It has the highest level of detail.

When performing the act of abstraction, software engineers attempt to create *procedural* and *data* abstractions. **Procedural abstraction** is abstraction applied to functions, methods, and procedures in general. For example, a function's name is an abstraction of the operations which the function performs. The name, in other words, stands for those sequence of operations, and in our designs can be used in its stead.

Data abstraction is abstraction applied to the data discussed in our software and system designs. When we combine data to form an object or class, or combine objects to form a collection, we are performing data abstraction.

Clearly, the software analysts and designers need to choose an appropriate level of abstraction to operate at.

Architecture

Architecture refers to the software being developed. Specifically, the **software's architecture** is the structure of the software: the components that make up the software and how these components are brought together.

Through abstraction, the software's architecture can also be represented at different levels of detail. At lower levels of abstraction, software architecture is concerned with classes, objects, and their interrelationships; at higher levels, the components we consider are the systems and subsystems which make up the software.

Note

The software's architecture is one of the most important aspects of a software system. As such, it should be considered early in the process of analysis and design. A poorly developed architecture is a high-risk factor in software development.

A number of models can be used to represent the software's architecture:

- *Structural models* represent the program's components (such as class diagrams).

- *Framework models* attempts to discover portions of the architecture that can be reused in similar programs. These reusable portions are called frameworks.

- *Dynamic models* show how the architecture may change over time, especially when the software needs to react to external events (such as sequence diagrams).

- *Process models* show the business / technical processes that the software captures (such as data-flow diagrams).

- *Functional models* shows the software's functional hierarchy.

Patterns

Patterns are known solutions to particular problems. Being known solutions, they can be useful guides to generating designs of new systems. Importantly, a design pattern should allow a software engineer to determine if the solution it specifies is suitable to solving a particular problem.

Design patterns are important enough to warrant their own chapter. We will discuss them in far more detail in Chapter 8, *Design Patterns*.

Modularity

When developing software, the software is broken into smaller and smaller components, into packages of classes, then into the classes themselves, into the base data-types that make up these classes, into the functions that they call, and so on.

This ability to divide a software system into discrete portions is called **modularity**, which is an important component of abstraction and architectural design. Having modular software allows it to be more easily comprehended by the developers and our customers. However, modularity has a drawback: while increasing modularity can increase our understanding of the software, after a certain point the software will consist of enough modules that we will again have a problem seeing how they all interact.

As with any abstraction tool, it is important to choose the right level of modularity (the right level of abstraction) for the software.

Modularised software is easier to develop and to test; it can more easily accommodate change, since change should be restricted to only a small number of modules.

Note

When we use the term *module*, we are referring to any division in the software, such as a package or a class.

Information hiding

Information hiding is an important aspect of modularity, and if you recall the definition of abstraction (*reducing information content to only what is important*), information hiding is an important aspect to the abstraction of software.

Specifically, consider that the final software system *is* the lowest level of abstraction. All of the software's design details are present at this level. Information hiding allows us to hide information unnecessary to a particular level of abstraction within the final software system, allowing for software engineers to better understand, develop and maintain the software.

We use software modules to implement information hiding: the information contained in the modules should be hidden from those the rest of the software system outside of the module, and access to this hidden information should be carefully controlled. This allows us to maintain a higher level of abstraction in our software, making our software more comprehensible.

If information hiding is done well, changes made to the hidden portions of a module should not affect anything outside of the module. This allows the software engineers to more readily manage change (including changes in the requirements).

Functional independence

Functional independence occurs where modules (such as a package or class) address a specific and constrained range of functionality. The modules provide interfaces *only* to this functionality. By constraining their functionality, the modules require the help of fewer other modules to carry out their functionality.

The functional independence of a module can be judged using two concepts: *cohesion* and *coupling*: **cohesion** is the degree to which a module performs only one function. **coupling** is the degree to which a module requires other modules to perform its function.

Note

The goal of functional independence is to maximise cohesion while minimising coupling.

Having many functionally independent modules helps a software system be resilient to change: because functionally independent modules rely on fewer other modules, there is less chance of changes to these modules spreading to those which are functionally independent.

Functional independence makes modules easier to develop and test. Changes made to how they perform their function are less likely to affect the software as a whole.

Functional independence is one of the goals of using information hiding and modularity. Consider this: there can be no good information hiding if the software has not been broken into modules. If the software has not been broken into modules, there can not ever be functionally independent modules. If no information is hid from other modules of the software, if every module always depended on all the others to perform its function, any change made to the software will always result in changes having to be made elsewhere in the software in order to handle these changes.

Stepwise refinement

Stepwise refinement is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and, incrementally refining the software through each level of abstraction, providing more detail at each increment. At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed.

At the early steps of the refinement process the software engineer does not necessarily know how the software will perform what it needs to do. This is determined at each successive refinement step, as the design and the software is elaborated upon.

Refinement can be seen as the compliment of abstraction. Abstraction is concerned with hiding lower levels of detail; it moves from lower to higher levels. Refinement is the movement from higher levels of detail to lower levels. Both concepts are necessary in developing software.

Refactoring

Refactoring is the activity of changing the software's internal structure without changing its external behaviour. It is specifically concerned with *improving* the software's internal structure.

Refactoring provides higher quality software, and eases the work of the software engineer. During refactoring, the software is examined for:

- *Redundant code*: portions of the software that perform the same function should be merged. Having the functionality repeated in multiple areas makes the software harder to maintain.

- *Unused design elements*: if portions of the software are not being used, and removing it will not change the software's behaviour, those portions should not be in the software.

- *Poorly constructed data structures*: these should be modified to improve, for instance, information hiding and functional independence.

As with all of the design concepts we have discussed, refactoring should be done to make code easier to understand, easier to develop and test, and easier to change. A good indication that you need to refactor a particular software application is when it has become difficult to either add new functionality to it, or to fix a bug in it.

Note

Refactoring is *not* concerned with fixing bugs or adding functionality: it is concerned with improving design concepts in the software. This, however, should make it easier to find and fix software bugs. Importantly, refactoring does *not* include changing the running time of an application: as with bug fixing or adding functionality, a change to the running time is clearly changing the software's behaviour, and so not in the purview of refactoring.

Design classes

As the design progresses, classes can often be fit into various roles. Five common roles are presented below.

- **User interface classes.** Instances of these classes are used to provide all the interaction between the user and the software. Often, interaction with the software occurs through the use of a metaphor (think of the desktop metaphor, or the drawing board metaphor in *Computer Aided Design* software), and user interface classes may represent elements of this metaphor.

- **Domain classes.** These are the classes that are used to implement some specific portion of the problem domain that the software is attempting to solve. Classes that represent books and catalogues are examples of domain classes for software used in a library.

- **Process classes.** Are lower-level domain classes used to implement the software.

- **Persistent classes.** Are classes that represent data stores, and data that will persist even when the program is not executing. They are useful for hiding the details of obtaining specific data from databases and files.

- **System classes.** Provide the functionality the software requires to operate and communicate with the environment in which it will be functioning.

When used in software design, these classes can be represented using appropriately named stereotypes. For example, user interface classes can be represented in class diagrams using the «user interface» stereotype, persistence classes with «persistent», and system classes with «system». However, the stereotypes should only be used if they will add useful meaning to the model. If knowing that a particular class will be used in the user interface is not useful, do not add the stereotype.

These classes should have the following properties:

- The class should do all that its name implies, and do only what its name implies.

- The class and each of its method should provide only one way to do the same thing.

- The class should be functionally independent. That is, it should have high cohesion and low coupling.

Review

Questions

Review Question 1

Why is the importance of *abstraction*?

Discussion of this question can be found at the end of this chapter.

Review Question 2

What is the main technique for implementing information hiding?

Discussion of this question can be found at the end of this chapter.

Review Question 3

Complete the following:

Refactoring is the activity of changing the software's [_____] without changing its [_____]. It is specifically concerned with improving the software's [_____].

During refactoring, software is examined for three things, namely:

1. [_____]

2. [_____]

3. [_____]

Discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Abstraction allows us to focus on the relevant portions of a problem and our design. It allows us to simplify our representation of how actions are carried out (procedures) and how data is represented (such as by using classes and objects).

Abstraction also allows us to represent our software at various levels of detail.

Discussion of Review Question 2

Software modules are the main tool used to implement information hiding. They segregate a portion of the information contained within a software system from the rest of the system, and control access to this information. Examples of software modules include C++ and C# namespaces, Java packages, and classes and objects.

Discussion of Review Question 3

Refactoring is the activity of changing the software's **internal structure** without changing its **external behaviour**. It is specifically concerned with improving the software's **internal structure**.

During refactoring, software is examined for three things, namely:

1. Redundant code

2. Unused design elements

3. Poorly constructed data structure

Chapter 8. Design Patterns

Objectives

At the end of this chapter you should be able to:

• Describe the provenance of design patterns and explain their potential use in the design process.

• Select a specific design pattern for the solution of a given design problem.

• Create a catalogue entry for a simple design pattern whose purpose and application is understood.

Introduction to design patterns

The idea of a pattern

A bridge is a structure used for traversing a chasm. In its basic form it consists of a beam constructed of rigid material, the two ends of the beam fixed at opposite ends of the chasm.

Figure 8.1. A simple pattern for a bridge
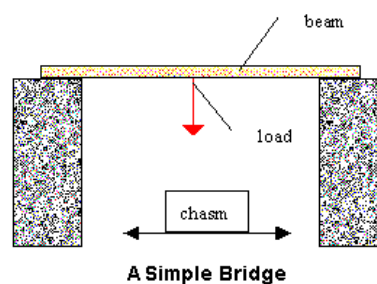
The bridge will fulfil its function if the rigidity of the beam can support the loads which traverse it. The beam's rigidity depends on the material of its construction and its span. In situations where the heaviness of the load, the length of the span or the material of construction are likely to lead to failure, the design of the bridge needs to be modified. More rigid materials are generally more expensive and/or more difficult to work with, so we shall ignore this possibility. This leaves two possible approaches:

1. Increasing the rigidity.

2. Decreasing the span.

Increasing the rigidity

The rigidity of the bridge structure can be improved by supporting the beam in a number of ways.

The girder

We can redistribute the beam's material to improve its rigidity.

Figure 8.2. The girder

A Girder Bridge
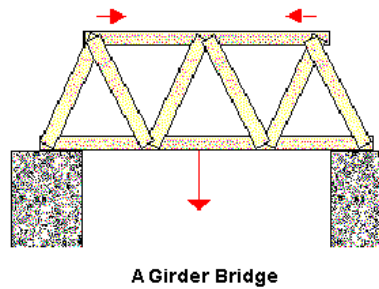
Some of the force of the load on the lower beam is distributed by the cross-members resulting in a compressive force in the upper beam. The (same) material of construction of the upper beam is better able to support compressive forces along its length.

The arch

Some of the force of the load on the beam is distributed compressively along the material of the arch.

Figure 8.3. The arch

An Arch Bridge
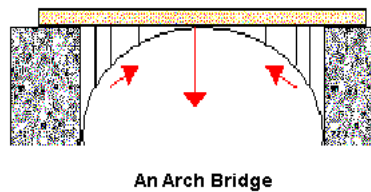
The arch must be specially shaped so that the forces remain compressive along the length of the arch. This shape is called a catenary.

Suspension

The arch can be replaced by a cable which supplies the same support from above rather than below.

Figure 8.4. Suspension

A Suspension Bridge
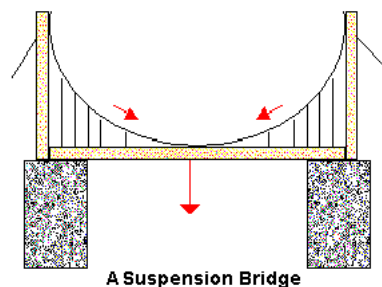
The cable hangs in a catenary shape.

Decreasing the span

If the chasm is not too deep it may be possible to divide it into two "sub-chasms" by building a support in the middle.

Figure 8.5. Subdivision

Sub-dividing the Chasm
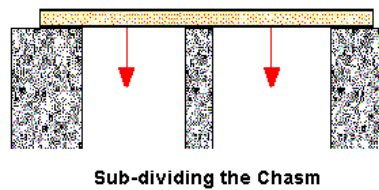
Alternatively, the edges of the chasm can be extended to reduce the length of the span.

Figure 8.6. Narrowing

Narrowing the Chasm
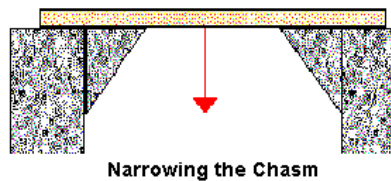
Some bridges are built by combining several of these approaches into and elegant and functional structure.

Figure 8.7. The Forth Bridge
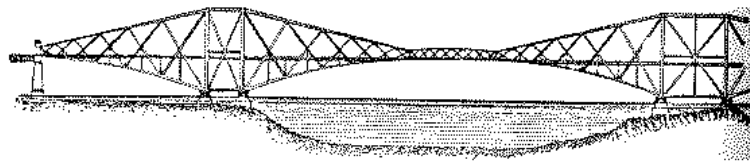
The Firth of Forth Bridge (taken from Encyclopedia Britannica, 1896)

The Forth Bridge is a railway bridge over the Firth of Forth river in Scotland.

Patterns

These are the *patterns* of bridge design. Even though we have no specialist knowledge of civil engineering, we can see how and why they work, and when we look at a bridge we can see the patterns which were used to design it because they are built into its structure. Civil engineers, whose work is to build such bridges, learn these patterns along with a great deal of specialised knowledge about construction materials and very sophisticated analytical techniques which help them to make precise predictions about the strength and suitability of different designs and materials before the bridges are built. They use the patterns, either singly or in combination, when thinking about and discussing the design of a particular bridge with other engineers, and they recognise the patterns used in constructing the bridges of their fellow engineers.

The origins of design patterns

The ideas behind design patterns come not from civil engineers but from architects. Like civil engineers, architects are concerned with designing structures to meet certain functional requirements. Individual buildings need to have a form that helps them to fulfil their purpose. For instance, a dwelling for a single family needs to have a living area, a sleeping area and utility rooms, like a kitchen and a bathroom. A good dwelling design will place these together in such a way that makes family living convenient and pleasant. The sleeping area should be away from the living area so that it is quiet. The bathroom will be directly accessible from many parts of the house, and not, for example, via one of the bedrooms. The kitchen should be accessible from the living area but not from the sleeping area. In

addition, the design will make efficient, effective and economic use of the materials of construction, for example roofing, materials and plumbing.

In their working lives architects "solve" these problems over and over again, in slightly different ways and under different circumstances. However, the basic precepts of good dwelling design remains the same, and they are different from the precepts of good office block design, or good hotel design or good hospital design. The design patterns of architects are the solutions they have found to all these various problems. Like civil engineers, architects can see the patterns in the buildings they design and see examples in the designs of other architects every time they walk into a house or drive down a street.

In the 1970's the architect Christopher Alexander wrote a series of books in which he first enunciated the idea of the design pattern, and the ways in which patterns could be used to solve specific architectural problems, and be combined to communicate design ideas amongst architects. He said:

> "Each pattern is a rule which describes what you have to do to generate the entity which it defines."
>
> — Christopher Alexander, *The Timeless Way of Building*

This definition gives a pattern two roles – firstly, as an abstract description of a solution to a particular type of problem; and secondly as a form which we can recognise within an "entity" which solves the problem. Thus a suspension cable is a way of providing support to the beam of a bridge, and when we look at a bridge we can see whether or not it uses a cable for support.

Patterns in software design

Many aspects of the work of software engineers have parallels with the work of civil engineers and architects. For instance:

• Software engineers design and construct software systems to meet certain functional requirements.

• These software systems may consist of a number of software components which must work together in a structure to deliver the functions.

• The designers must concern themselves with effective, efficient and economic construction and operation.

• There are precepts of good design for various types of software system.

• Software engineers solve variations of particular design problems over and over again.

However, there are also some stark differences.

• Although they may use other engineers' software, software engineers rarely have the opportunity to observe other people's software designs in the same intimate way that an architect can when he or she enters a building.

• Both civil engineering and architecture are very old disciplines stretching back thousands of years. In contrast, software engineering is a new discipline, only a few decades old.

It is not surprising that software engineers would look to more mature design disciplines for some assistance in defining what they were trying to do and in a search for techniques to help them to do it. As explained in Chapter 2, *Process and Model*, the term *software engineering* itself was coined to emphasise the belief that it took more than just skillful programming to produce a good piece of software, and that careful consideration of requirements combined with systematic design and development would help to bring software artefacts up to the same levels of reliability and elegance as well-engineered "hardware".

The work of Alexander was known to two software engineers, Ward Cunningham and Kent Beck, when, in 1987, they visited a client to discuss the design of a user interface. They liked Alexander's idea that a pattern is a symbolic way of describing a solution to a type of problem, and that a set of patterns could provide a language for discussing the problem and considering various solutions. They wanted the users of the proposed system to contribute to the design (another Alexandrian precept), so

they invented a small set of user interface patterns for their users. They became convinced of the value of these ideas and their relevance to software engineering when the users produced a very elegant and efficient design using the simple pattern language.

Ward and Cunningham presented their conclusions at a software engineering conference (OOPSLA '87) but few of the delegates were convinced. However, at about the same time, other workers in software engineering were feeling their way towards an architectural view of software engineering and by the 1991 OOPSLA conference the term "design patterns" was in use. A community of software engineers was gradually developing who were taking Alexander's work very seriously. Many of the leading members of this community attended a meeting in 1993. During the meeting, they decided to try designing a building according to Alexander's principles. This included laying out the physical building plan, which they duly did on the side of the hill in Colorado where the meeting was being held. Henceforth, they were known as the *Hillside group*.

Finally, in 1995, four of the main proponents of design patterns, E. Gamma, R. Helm, R. Johnson and J. Vlissides published the first book on design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software*. This is still perhaps the most authoritative book on the subject and much of the ensuing material is drawn from there. The four authors are known in the design pattern community as "The Gang of Four".

Design patterns in object-oriented programming

Definitions of terms and concepts

The following is a summary of terms you were already introduced to in the earlier chapters that will be essential for the understanding of design patterns.

Object

One of the main tasks of object-oriented design is to identify the classes which make up the software system (see Chapter 5, *Object-oriented Analysis and Design*).

Not all objects that will be part of a system are identified early on in the development process, for a number of reasons, including the chosen software process (such as incremental processes).

Interface

The most important aspect of an object is its *interface*. An object's **interface** defines how the object can be used, in other words, to what kind of messages it can respond. The parameters that need to be passed with the message, if any, and the return type are called collectively the operation's signature. The implementation details of these operations do not need to be known to the client.

Many operations with the same name can have different signatures, and many operations with the same signature can have different implementations (using inheritance). These are forms of *polymorphism*. This substitutablity — in other words, being able to substitute objects at execution time — is called **dynamic binding**, and is one of the main characteristics of object-oriented software. Objects with identical sets of signatures are said to conform to a common interface.

Class

A class definition can be used as a basis for defining subclasses by means of inheritance. A subclass possesses all the data and method implementations of the superclass together with additional data and methods pertaining exclusively to objects of the subclass. In some cases, subclass data may shadow superclass data with the same identifiers, or may override methods with the same signature. An abstract class is a class that can have no objects. Its main purpose is to define a common interface shared by its subclasses. Sub-classes specify implementations for these the methods of an abstract class by overriding them.

There is a distinction between *inheritance* and *conformance*. In Java, this is explicitly defined by means of extending a class through inheritance, and by implementing an interface to ensure conformance to certain behaviour. An object's type is defined by its interfaces; this defines the messages to which it can respond or, in other words, how it can be used. A class is a type, but objects of many different classes can have the same type.

Scope of development activity: applications, toolkits, frameworks

Software developers may find themselves involved in different sorts of software development activities. Most developers work on applications designed to be used by non-specialist computer users to perform tasks relevant to their particular work. However, some developers may be involved in producing specialist software designed to help application software developers in the production of their applications. The products of such developers are variously called *toolkits* or *frameworks*, depending on the scope of their applicability.

When developing an application it is necessary to consider reusing existing software, as well as making sure the newly developed software is easy to maintain and is itself reusable. Maintenance is in itself a form of software reuse.

The smallest unit of reuse in object-oriented software is an object or class. When a class is reused (e.g., refined by means of sub-classing) this is called **white-box reuse**. This is due to visibility: all attributes and methods are normally visible to sub-classes. This type of reuse is considered to be more complex for developers, because it requires an understanding of the implementation details of the existing software. When reuse is by means of object composition, and we are only concerned with the interfaces – how an object can be used — this is called **black-box reuse**, because the internal details of the object are not visible.

Black-box reuse has proved to be much more successful than white-box reuse. It is less complex for developers and does not interfere with the encapsulation of objects and is therefore safer to use.

Toolkits are a set of related and reusable classes designed to provide a general purpose functionality. Toolkits help with the development process without imposing too many restrictions on the design. The packages in Java such as java.net, java.util, and the java.awt are examples.

Frameworks represent reuse at a much higher level. Frameworks represent design reuse and are partially completed software systems intended for a specific family of applications. One example of a framework is the *Java Collections Framework*.

Patterns, in contrast, are not pieces of software at all. They are more abstract, intended to be used for many types of applications. A **pattern** is a small collection of objects or object classes that co-operate to achieve some desired goal. Each design pattern concentrates on some aspect of a problem and most systems may incorporate many different patterns.

Pattern classifications and pattern catalogue

Design patterns are based on practical solutions that have been successfully implemented over and over again. Design patterns represent a means of transition from analysis/design to design/implementation.

To help developers to use design patterns, catalogues of patterns have been created. Each catalogue entry for a pattern should contain the following four essential elements:

- *The pattern name*, which identifies a commonly agreed meaning and represents part of the design vocabulary.

- The *problem* or *family of problems* and *conditions* to which it may be applied.

- The *solution*, which is a general description of participating classes/objects and interfaces their roles and collaborations.

- The *consequences* — each pattern highlights some aspect of the system and not others, so it is useful to be able to analyse benefits and restrictions.

Gamma et al classify design patterns into three categories according to purpose. The categories are *behavioural*, *creational* and *structural*. Unfortunately the catalogue of patterns is not standardised, which may cause some confusion. The level of granularity and abstraction differs greatly from objects whose only responsibility is to create other objects to those that create entire applications. There is no guarantee that a suitable pattern will always be found. It also may be that several different patterns could be used to solve a specific problem — in other words, a single pattern may not represent the only solution, but a possible solution.

Table 8.1. Design patterns according to Gamma et. al.

Behavioural	Creational	Structural
Interpreter	Factory Method	Adaptor (class)
Template Method	Abstract Factory	Adaptor (object)
Chain of Responsibility	Builder	Bridge
Command	Prototype	Composite
Iterator	Singleton	Decorator
Mediator		Facade
Momento		Flyweight
Observer		Proxy
State		
Strategy		
Visitor		

The Portland Pattern Repository

A large collection of design patterns is available at the Portland Pattern Repository [http://c2.com/ppr/]. This repository is hosted by *Cunningham & Cunningham*, the consultancy firm of Ward Cunningham, one of the Gang of Four. The website is also famous for being the web's first wiki.

Behavioural patterns

Behavioural patterns are required when the operations that need to be performed cannot be achieved without co-operation. Thus behavioural patterns concentrate on the way in which classes and objects organise responsibilities in order to achieve the required interaction.

The *Observer* pattern is an example of a behavioural pattern that defines some dependency between objects.

The *Observer* pattern

In some applications, two or more objects that are independent of each other must respond to some event in synchrony. For example any *Graphical User Interface* (GUI) will respond to the click of a mouse button, or keyboard, which will trigger the execution of an application or utility and redraw the screen appropriately. The mouse click event will result in one or more objects responding. Each object is otherwise independent. Each object is able to respond only to certain events.

Other typical examples of applications in which the *Observer* pattern could be used:

- To integrate tools in a programming environment. For instance, the editor for creating program code may register with the compiler for syntax errors. When the compiler encounters such an error, the editor will be informed and can scroll to the appropriate line of code.

- To ensure consistency constraints, such as referential integrity for database management systems.

- In the design of user interfaces to separate presentation of data from applications that manage the data. For example a spreadsheet object and a chart or a text report may all display the same information resulting from an application's data at the same time in their different forms.

The problem

The important relationship that needs to be established is between a *subject* and an *observer*. The subject may be observed by any number of observers. The observers should be notified when the subject changes state. Each observer will receive information concerning the subject's state with which to synchronise, and to allow them to respond as required by the application. The following summarises the conditions:

- The subject is independent, and the observers are dependent.

- A change in the subject will trigger changes in observers — of which there may be many.

- The objects that will be notified by the subject are otherwise independent. They only share in some aspect of their behaviour.

Figure 8.8. Class diagram of the *Observer* pattern
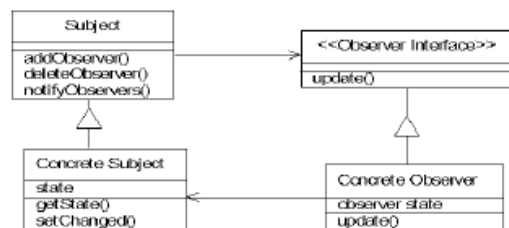
The solution

Participating classes /objects:

In the diagram the subject is shown as a class. *Subject* has methods for attaching and detaching observer objects. The methods are shown on the diagram as addObserver(), deleteObserver() and notifyObservers().

Observer has an updating interface for objects that will be notified of changes in a subject, here shown as an interface with the method update().

Concrete Subject, a subclass of *Subject*, contains its state (of interest to the observers), plus operations for changing its state. *Concrete Subject* is able to notify its observers when its state changes. On the diagram the attribute state, and methods getState(), setChanged() provide this functionality, and the other methods are inherited from *Subject*.

Concrete Observer maintains a reference to the *Concrete Subject* object, and a state that needs to be kept consistent with the *Concrete Subject*. It implements the updating interface. On the diagram the *Concrete Observer* is a class that implements the *Observer* interface. It supplies the code for the update() method and has observerState to denote the data that needs to be kept consistent with the *Concrete Subject* object.

Collaborations

The *Concrete Observers* objects register with the *Concrete Subject* object, using the addObserver() method.

When a *Concrete Subject* changes state it notifies the *Concrete Observer* objects by executing the notifyObservers() method.

The *Concrete Observer* object(s) obtain the information about the changed state of the *Concrete Subject* and execute the update() method.

Consequences

The advantage of the pattern is that the *Subject* and *Observer* are independent of each other, and the subject does not need to know anything about the handling of the notification by the observers (i.e., how update() works). This means that any type of broadcasting communication could be implemented in this way.

Creational patterns

Creational patterns handle the process of object creation. These patterns may be used to provide for more reusable designs by placing the emphasis on the interfaces and not the implementation. The abstract factory is an example of a creational pattern that can be used to make objects more adaptable, in other words:

• Less dependent on specific implementations.

• More amenable to change and customisation, easier to change the objects themselves.

• Less necessary to change the applications that use the objects.

Abstract factory pattern

The abstract factory pattern makes the system independent of how objects are created, composed and represented. It should be used whenever the type or specific details of the actual objects that need to be created cannot be predicted in advance, and therefore must be determined dynamically. Only the required behaviour of the objects is specified in advance. The information that can be used to create the object would be based on data passed at execution time. Examples of applications of the pattern:

• To customise Windows, Fonts, and so on, for the platform on which the application will run so as to ensure appropriate formatting, wherever the application is deployed.

• When the application specifies all the required operations on the objects it will use, but their actual format will be determined dynamically.

• To internationalise user interfaces (e.g., to display all the text in a local language, to customise the date format, to use local monetary symbols).

The problem

The application should be independent of how its objects are created and represented. It should be possible to configure the application for different products/platforms. The application defines precisely how the objects are used (i.e., their interfaces).

The solution

Participating classes/objects

Abstract Factory class will contain the definition of the operations required to create the objects, createProductA(), createProductB().

Concrete Factory implements the operations createProductA(), createProductB(). Only one *Concrete Factory* is created at run time which will be used to create the product objects

AbstractProduct will declare an interface for the type of product object for example a particular type of GUI object: Label or Window.

ProductA will define the object created by the *Concrete Factory 1*, implementing the *Abstract ProductA* interface.

Client Application uses only the interfaces from the *Abstract Factory* and *Abstract Product* classes.

Figure 8.9. Class diagram for the *Abstract Factory* pattern
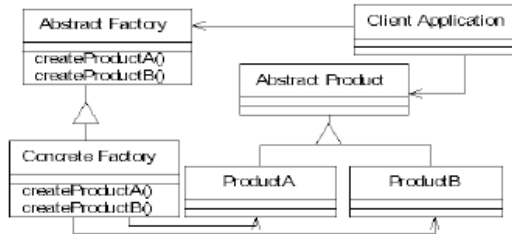
Consequences

Different product configurations can be used by replacing the *Concrete Factory* an application uses. This is a benefit and liability, because for each platform or family of products a new *Concrete Factory* subclass needs to be defined. However, the changes will be broadly restricted to the definition of subclasses of *Abstract Factory* and *Abstract Product*, thus confining the changes to the software to well documented locations.

Structural patterns

These patterns deal with the composition of complex objects. Similar functionality can be often achieved by using delegation and composition instead of inheritance. An example of a structural pattern is the composite pattern. In the Java API, this pattern is used to organise the GUI using AWT objects and layout managers.

Composite pattern

The pattern involves the creation of complex objects from simple parts using inheritance and aggregation relationships to form treelike hierarchies.

The diagram shows the composite pattern as a recursive structure where a component can be either a leaf (which has no sub-components of its own) or a composite (which can have any number of child components). The component class defines a uniform interface through which clients can access and manipulate composite structures. In the diagram this is represented by the abstract method operation().

Figure 8.10. Class diagram of the *Composite* pattern
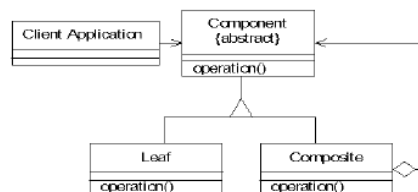
The problem

Complex objects need to be created, but the composition and its constituent parts should be treated uniformly.

The solution

Participating classes/objects

Component should declare the interface for the objects in the composition, as well as interfaces for accessing and managing its child components.

Leaf represents objects that have no children, and defines behaviour for itself only.

Composite will define behaviour for components with children, and implements the child related interfaces.

Client Application manipulates objects through the component interface using the operation() method. If the object to be manipulated is a *Leaf* it will be handled directly, if it is a *Composite*, the request will be forwarded to the child

Consequences

The pattern enables uniform interaction with objects in a composite structure through the *Component* class.

Defines hierarchies consisting of simple objects and composite objects which can themselves be composed and so on.

Makes it easier to add or remove components.

How to use a design pattern

- Consult design pattern catalogues for information (such as the Portland Pattern Repository, discussed earlier). You may find an example or description that may suggest the pattern is worth considering.

- Try to study the suggested solution in terms of participating objects/classes, conditions, and descriptions of the collaborations.

- If the examples of these patterns are part of a toolkit, it may be useful to examine the available information. java.util supports the *Observer* pattern, for example.

- Give participant objects names appropriate for your application context.

- Draw a class diagram showing the classes, their necessary relationships, operations and variables that are needed for the pattern to work.

- Modify the names for the operations and variables appropriately for your application.

- Try out the pattern by testing a skeleton example.

- If successful refine and implement it.

- Consider alternative solutions.

Patterns in Java

Some design patterns generally recognised as common solutions to specific problems have been adopted as part of the Java JDK and Java API. A sample of these design patterns will be analysed here in greater detail.

The *Observer* pattern in Java

In Java, the *Observer* pattern is embodied by the *Observer* interface and the *Observable* class which are part of the java.util package.

Any object that needs to send a notification to other objects should be sub-classed from class *Observable*, and any objects that need to receive such notifications must implement the interface *Observer*.

Table 8.2. *Observable* class methods in java.util.package

Observable Methods	Description
addObserver(Observer o)	Add the object passed as an argument to the internal record of observers. Only observer objects in the internal record will be notified when a change in the observable object occurs.
deleteObserver(Observer o)	Deletes the object passed as an argument form the internal record of observers.
deleteObservers()	Deletes all observers from the internal records of observers.
notifyObservers(Object arg)	Call the update() method for all the observer objects in the internal record if the current object has been set as changed. The current object is set as changed by calling the setChanged() method. The current object and the argument passed to the notifyObservers() method will be passed to the update() method for each *Observer* object.
notifyObservers()	Same but with null argument.
countObservers()	The count of the number of observer object for the current object returned as an int.
setChanged()	Sets the current object as changed. This method must be called before calling the notifyObservers() method.
hasChanged()	Returns true if the object has been set as "changed" and false otherwise.
clearChanged()	Reset the changed status of the current object to unchanged.

addObserver() method of the *Observable* class registers the *Observers*.

Each class that implements an *Observer* interface will have to have an update() method, and this method will ensure that the objects will respond to the notification of a change in the *Observable* object by executing the update() method:

```
public void update(Observable, Object)
```

The *Observable* object can indicate that it has changed, by invoking at any time notifyObservers()

Each *Observer* is then passed the message update() where the first argument is the *Observable* that has changed and the second is an optional one provided by the notification.

Example of *Observer* pattern using java.util

An electrocardiogram (ECG) monitor attached to a patient notifies four different devices:

- *Remote Display* for the physician, to allow them to adjust the configuration settings and treatment.

- *Chart Recorder* that will display the waveforms.

- *Remote Display* with a patient alarm.

- *Instruments Monitor* for the service personnel.

Figure 8.11. The ECG *Observer*
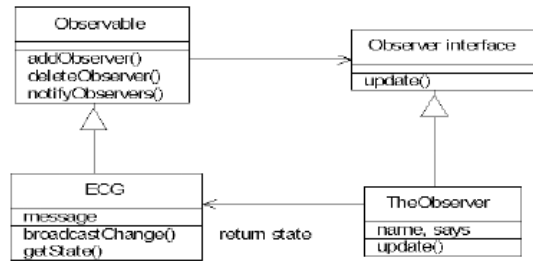
Each of the devices will perform specific operations when the events for which they have registered an interest take place. The ECG attached to the patient will notify them of changes, as they occur. The skeleton solution presented below will display messages stating the specific tasks performed by each *Observer*.

```
/*========================================================== ECG.java


Definition of class ECG subclassed from Observable
from which Observable objects will be created.
Demonstrates the use of methods setChanged() which will change
the state of the ECG object, and notifyObservers() which will
broadcast the event to the registered objects.

==============================================================*/

import java.util.*;

public class ECG extends Observable
{
    String message = "";

    public void broadcastChange()
    {
        message = "\tHeart Electrical Activity";
        setChanged();
        notifyObservers();
    }

    public String getState()
    {
        return message;
    }
}

//class




/*==========================================================

TheObserver.java


Definition of class TheObserver from which Observer objects will
be created.

The class implements the Observer interface and thus must define
```

```
the method update() which will be executed when the objects of
class TheObserver are notified.

================================================================*/

import java.util.*;
import java.io.*;

public class TheObserver implements Observer
{
  String name;
  String says;

  public TheObserver (String name, String says)
    {
        this.name = name;
        this.says = says;
    }

    public void update(Observable O, Object o)
    {
        System.out.println(((ECG)O).getState()+
        "\n\t" + name + " : " + says+ "\n");
    }
}

//class


/*============================================================

PatternTest.java

Definition of program to test the Observer pattern

An object of class ECG called ecg is created, as well an array
called observers containing the four TheObserver objects.

These objects register with the observable object ecg using method
addObserver.

When the ecg method broadcastChange() is executed the observers
will be notified and update themselves.

================================================================*/

import java.util.*;

public class PatternTest {

    public static void main(String[] args) {
        ECG ecg = new ECG();
        TheObserver[] observers
        = { new TheObserver("Physician", "Adjust Configuration"),
            new TheObserver("Remote Display","Monitor Details"),
            new TheObserver("Chart Recorder", "Draw ECG WaveForm")
            new TheObserver("Service Personnel", "Monitor Instruments")};
```

```
        for (int i = 0; i &lt; observers.length; i++)
            ecg.addObserver(observers[i]);

        ecg.broadcastChange();
    }
}
```

The *Observer* pattern as part of the Java API

An example of the application of the *Observer* pattern of the Java API is the Java model of event handling using listeners. *Graphical User Interface* (GUI) components from the Java *Abstract Windowing Toolkit* (AWT) such as buttons, text fields, sliders, check boxes, and so on, are managed in this way. The observer objects implement a listener interface (e.g., the ActionListener, WindowListener, KeyListener etc.). When any of the components changes state, the listeners are notified that a change has occurred. The listener decides what action should be taken as a result of the change. To tell a button how the events it generates should be responded to, the button's addActionListener() method is called, passing a reference to the desired listener. Every component in AWT has one addXXXListener() method for each event type that the component generates. When a button with an action listener is clicked, the listener receives an actionPerformed() call which will contain the instructions that need to be performed in response to the event. The actionPerformed() method must be defined as part of implementing the listener interface

The *Model-View-Controller* pattern

This pattern is a specialised version of the *Observer* pattern, which was introduced in the *Smalltalk* language as a way of structuring GUI applications by separating the responsibilities of the objects in the system. The user interface consists of a *View*, which defines how the system should present this information to the user, and a *Controller*, which defines how the user interacts with the *Model*, which receives and processes input events. Systems analysis and design concentrates mainly on building the model representing the main classes of the application domain and the information of interest will be internally stored in object of the classes. The *Model-View-Controller* pattern makes it possible to associate the model with many different view/controller pairs in a non-intrusive manner without cluttering the application code. It is introduced here because it can be easily applied to the way in which Java applications using the java.awt can be structured. It provides a way for providing the application system model with a user interface. This is achieved by separating the responsibilities as follows:

The responsibilities of the *Model* are:

• To provide methods that enable it to be controlled.

• To provide a method or methods to update itself, and if it is a graphical object, to display itself.

The *Controller* carries out the following series of actions:

• The user causes an event such as clicking the mouse over a button.

• The event is detected.

• A method to handle the event is invoked.

• A message will be sent to the model.

• The update method within the model will change the data within the model.

• A message will be sent to the view update the user interface, to, for example, redraw the image.

The *View* performs the following:

• Initially displays the model when the window is created, and every time it is resized.

- When the event handler detects a change (e.g., responds to a click of a button) it sends a message to redraw the screen.

- The *View* enables the updated information from the model to be displayed.

Abstract factory facilities in Java

To make the creational process more versatile, object-oriented language facilities that provide for customisation should be used. The JDK has facilities to customise graphics and to internationalise programs and applications.

Graphics related platform characteristics

One of the problems of having a portable *Abstract Windowing Toolkit* is that the appearance and positioning of the objects may differ from one platform to another. To ensure that the graphical display is similar wherever the application may be ported, we should be able to adjust the *View* accordingly.

The *ToolKit* and *FontMetrics* classes may be used to help create a *Concrete Factory* that will create the *Concrete Product*:

Figure 8.12. Java output of screen and font information

The *ToolKit* class provides information regarding screen resolution, pixels, available fonts, and *FontMetrics* can help supply information concerning font measurements for every AWT component. Below is a program listing showing how some screen and font information can be obtained. Figure 8.12, "Java output of screen and font information", shows an example of some output obtained from the program.

```
/*=============================================================

Sysinfo.java

program for finding out details of the display using the
java.awt Toolkit class

The program creates a Toolkit class object theKit, and then uses
the Toolkit methods:

getDefaultToolkit, getScreenResolution, getScreenSize and getFontList.

=============================================================*/

import java.awt.*;

public class Sysinfo
{
    public static void main(String[] args)
    {
        Toolkit theKit = Toolkit.getDefaultToolkit();
        System.out.println("\nScreen Resolution: " +
```

```
    theKit.getScreenResolution() + " dots per inch");
Dimension screenDim = theKit.getScreenSize();

System.out.println("Screen Size: " + screenDim.width
    + " by " + screenDim.height + " pixels ");

String myFonts[] = theKit.getFontList();

System.out.println("\nFonts available on this platform: ");

for (int i = 0; i &lt; myFonts.length; i++)
    System.out.println(myFonts[i]);

        return;
    }
}
```
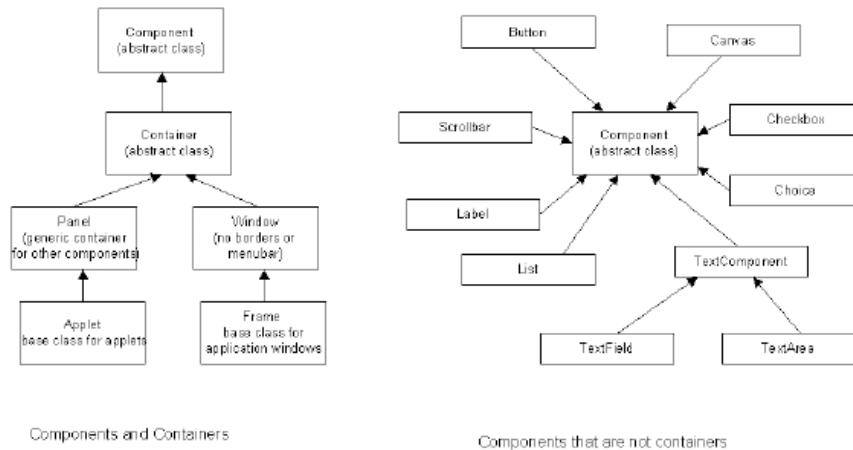
Java facilities for internationalisation of applications (displaying all user visible text in the local language, using local customs regarding dates, monetary displays etc.) are available from java.util package using the *ResourceBundle* class and its subclasses.

Composite patterns in Java

HCI classes for creating GUI applications are used in almost every interactive application. Much of these GUI classes adhere to some form of *Composite* pattern, and the Java AWT is no exception. Here, containers are components which can hold other components. Each component knows how to draw itself on the screen; containers, however, will defer some of their drawing functionality to the components that they contain.

Figure 8.13. java.awt GUI components containers and layout managers

Components and Containers

Components that are not containers

Layout Managers	Description
FlowLayout	Places components in successive rows in a container, fitting as many on each row as possible, and starting on the next row as soon as a row is full. This works in much the same way as a text processor placing words on a line. Its primary use is for arranging buttons, although it can be used with components. It is the default layout manger for *Panel* and *Applet* objects

Layout Managers	Description
BorderLayout	Places components against any of the four borders of the borders of the container and in the centre. The component in the centre fills the available space. This layout manger is the default for objects of the *Window*, *Frame*, *Dialog*, and *FileDialog* classes
CardLayout	Places components in a container, one on top of the other – like a deck of cards. Only the "top" component is visible at any one time.
GradLayout	Places components in the container in a rectangular grid with the number of rows and columns that you specify.
GridBagLayout	This places the components into an arrangement of rows and columns, but the rows and columns can vary in length. This is a complicated layout manager with a lot of flexibility for controlling where components are placed in a container.

In Java, these are embodied by the AWT *Component* and *Container* class, and the layout manager classes. A *Window* can be divided into *Panels*, and each *Panel* can be treated as an individual component within another layout at a higher level.

Review

Questions

Review Question 1

How many of the different methods of managing heavy loads have been used in constructing the Firth of Forth Bridge?

A discussion of this question can be found at the end of this chapter.

Review Question 2

What is the principal difference between the job of a software engineer and those of architects and civil engineers?

A discussion of this question can be found at the end of this chapter.

Review Question 3

Explain the role of design patterns in object-oriented software development.

A discussion of this question can be found at the end of this chapter.

Review Question 4

Place each of the following patterns in the category it belongs to according to "the gang of four":

Patterns: Observer, Model-View-Controlled, Abstract Factory, Composite:

Category: Creational, Structural, Behavioural.

A discussion of this question can be found at the end of this chapter.

Review Question 5

What are the four essential elements of a design pattern catalogue entry?

A discussion of this question can be found at the end of this chapter.

Review Question 6

What is meant by granularity?

A discussion of this question can be found at the end of this chapter.

Review Question 7

Give examples of white-box reuse and black-box reuse from the pattern examples.

A discussion of this question can be found at the end of this chapter.

Review Question 8

Compile ECG.java, TheObserver.java and TestPattern.java, then execute the TestPattern program.

1. Create a new directory in which to store the files. You may call the directory Activity1.

2. Copy all three files ECG.java, TheObservers.java and TestPattern.java

3. Compile the three files using the JDK command javac

4. Once the compilation is successful you should be able to execute the TestPattern application using the command java Testpattern

Does the program output the messages in the order you have expected?

A discussion of this question can be found at the end of this chapter.

Review Question 9

Design a solution using the Observer pattern for the operation of dispensing cash by an ATM machine. When a bank customer withdraws money from an ATM (Automatic Teller Machine), before the cash is dispensed it is necessary to determine whether there are sufficient funds. If there are, then it is necessary to instruct the machine to dispense the cash, to debit the customer's account and to log the transaction for auditing purposes.

How would you use the Observer pattern to design a solution to this problem?

A discussion of this question can be found at the end of this chapter.

Review Question 10

Draw a diagram to represent the design pattern of a solution to the ATM operation of dispensing cash using the Observer pattern. You may use a CASE tool for the class diagram and include the outline of the methods, data and messages required to make the pattern work. It would be useful if you put the project in which the class diagram will be placed into a new directory.

A discussion of this question can be found at the end of this chapter.

Review Question 11

The code you are expected to write will be a skeleton for the Observer pattern with messages instead of complex code to implement the operations. Compile the three parts of the program in the correct

order. The Observable class, then the Observer and last the program. You may use any version of Java for this exercise. Follow the instructions similar to the ones given for Activity 1. It would be useful to place all of your files for this exercise in a new directory.

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

The Firth of Forth bridge uses three patterns which can be directly seen. These are:

• girders with cross-members,

• arches in a catenary shape,

• decreasing the span.

In addition some suspension support is provided 'from above' but this is not strictly a catenary shape.

Discussion of Review Question 2

The main difference is that software engineers do not have the opportunity to see their and other people's designs implemented visually.

Discussion of Review Question 3

Your answer is expected to include some of the following:

Serve as exemplars to programmers, designers and architects, which they can quickly adapt for use in their projects.

Emphasise solutions: discovering patterns that have been used before rather than inventing them.

Represent codified, distilled wisdom: solutions to recurring problems, if those solutions have well understood properties.

Allow programmers and designers to program and design using bigger chunks; this also eases those aspects that involve understanding an architecture; architectural reviews. Reverse engineering, maintenance and system restructuring.

Aid in communicating among designers, between designers and programmers, and between a project's team members and its non-technical members.

Identify and name abstract, common themes in object-oriented design, themes that have known qualify properties.

Form a documented, reusable base of experience, which would otherwise be learnt only through an informal oral tradition or through trial and error.

Provide a target for reorganisation of software because a designer can attempt to map parts of an existing system to a set of patterns. If this mapping can be done, the complexity of the resulting reorganised system will be less than the original version.

Discussion of Review Question 4

Behavioural: Observer, M-V-C

Creational: Abstract Factory

Structural: Proxy, Composite

Discussion of Review Question 5

- The pattern name which identifies a commonly agreed meaning and represents part of the design vocabulary.

- The problem or family of problems and conditions in which it may be applied.

- The solution which is a general description of participating classes/objects and interfaces their roles and collaborations.

- The consequences - each pattern highlights some aspect of the system, and not others so it is useful to be able to analyse benefits and restrictions.

Discussion of Review Question 6

Granularity usually refers to the size of the components you deal with. In this context it could be from patterns that specify how a single object may be created, to patterns that will specify the structure of a whole application.

Discussion of Review Question 7

For example, in the Observer pattern:

Concrete Observer is an example of black-box reuse because all the Concrete observer needs to do is to implement the update() method.

The Concrete Subject is an example to white-box reuse, because it needs to know the details of its super class Observable.

The Concrete Factory of the Abstract Factory pattern is an example of white box reuse. The way in which the Client Application uses the Abstract Factory pattern is an example of black box reuse. Etc.

Discussion of Review Question 8

Compare this example with the Observer catalogue entry description, and follow the instructions on how to use a design pattern before going on to do the next exercise.

Discussion of Review Question 9

This exercise will help you test your understanding of the Observer pattern. You are expected to base your answer on the given examples – to literally use the pattern. Using the CASE tool will help you produce the documentation of your design in the standard format.

Discussion of Review Question 10

The purpose of this exercise is to use the Observer class and Observable interface from java.util and to test some of the methods from the lecture notes. This will consolidate your understanding of the use of this design pattern.

Discussion of Review Question 11

Writing portable software, and implementing dynamic binding of objects in a distributed environment for which Java was designed requires developers to be aware of the different platforms, and be able to make sure applications take advantage of the facilities in the language that make it possible to port.

Chapter 9. Software Testing

Objectives

At the end of this chapter you have gained an understanding of:

- What software bugs are.

- Who tests the software.

- How to write testable software.

- Various testing strategies, including unit testing and regression testing.

- Debugging.

Introduction to software testing

Software never runs as we want it to: creating software is an attempt at formally specifying (using a programming language) a solution to a problem. Assuming that it implements the correct solution (as defined by the by the various requirements and design documentation), it very often implements this solution incorrectly. Even if it does implement the solution correctly, the solution may itself not be what the customer wants, or may have unforeseen consequences in the software's behaviour. In all of these situations the software may produce unexpected and unintended behaviour. The error which causes this unexpected and unintended behaviour is called a **software bug**.

Testing is the planned process of running software with the *intent* of discovering errors in the software. It is important to realise that testing is a purposeful process, and is not the accidental discovery of software bugs.

Discovering errors in the software implementation itself is an important aspect to software testing, but it is not the only aspect. Testing can also be used to discover how well the software conforms to the software requirements, including performance requirements, usability requirements, and so on. Understanding how to test software in a methodical manner is a fundamental skill required in engineering software of acceptable quality.

This chapter considers various aspects of software testing.

The testers

Many different people involved with the development of a particular piece of software can be involved in its testing, such as the *developers*, a team of *independent testers*, and even the *customers* themselves.

The developers

Testing will always begin with the developers. As they develop individual modules of the software they will need to determine that what they have developed conforms with the requirements, and that it functions as expected. Further, the developers will have to test that these modules still function without error when they have been integrated with each other.

However, the developers will often test the software far more gently than is needed for proper testing, since the developers may already have pre-conceived expectations concerning how the software they have written behaves. Even worse, the developers may even have a vested interest in showing that

the work they have done performs correctly and meets the requirements; this works directly against the process of software testing.

An independent testing team

After the developer has produced working code, the code may be passed to an independent testing team. This is a group of developers who are *not* responsible for the original development of the software. This group's sole responsibility is to test the software that they have been given.

Independent testing teams are one solution to the problems that arise from having the original programmers also test all aspects of the software. Since the testing team did not develop the software, they will hopefully have less interest in reporting that the software functions better than it does. Also, teams dedicated to testing software can use more specialised testing frameworks and methodologies than the developers themselves. The original programmers may not be interested in testing the usability of the software, or may not have the resources to examine the performance of the software; the testing team, however, should have these resources.

The presence of testing teams does not mean that the original developers are not involved with the testing at all. The will still test their code as they develop it, and will be involved with the testing team as the team examines the software and reports on any errors in the software which they locate. The developer is usually the person responsible for correcting these errors.

The customer

During iterative and evolutionary development (and agile processes in general), software is frequently given to the customer for them to examine, report back on, and potentially use.

When the customer is closely involved with the software development, it is possible to have the customer perform limited testing of the software themselves. This is known as *beta testing*.

While the customer will not usually be able to test the software as thoroughly as the original software engineers, they will be able to examine how well the software meets their needs, whether the software requirements have to be changed, and whether there are any obvious bugs which the developers have missed.

Using beta testing is *not* a substitute for testing by the developers.

Principles of software testing

The completion of software testing

Software testing never completes. It is an ongoing process that begins at the project's inception and continues until the project is no longer supported. During the software's lifetime the burden of testing slowly shifts: from the developers during design and programming, to independent testing teams, and finally to the customers. Every action that the customer performs with the software can be considered a test of the software itself.

Unfortunately, time and money constraints often intervene: it may not be worth the developer's time or money to fix particular software errors. This trade-off between resources spent vs potential benefit can easily occur for small errors, and for errors which are not often encountered by the software's users.

It is also possible (although difficult) to be statistically rigorous when discussing software errors. For instance, it is possible to develop a statistical model of the number of expected software failures with respect to execution time. Error rates over a given period can then be specified given a particular probability. When that probability is low enough, testing could be considered "complete".

Writing testable software

An important aspect of testing is to ensure that the written software is written in a way that it can easily be tested. As the software becomes more difficult to test, so the software will be tested less often.

There are a number of guidelines that software engineers can follow in order to write software that can be easily tested. The design principles mentioned in Chapter 7, *Design*, are a good place to start. In addition to this, here are some further guidelines:

- **Operability**: This is partly a self-fulfilling quality: the fewer bugs that a software system has, the easier the software will be to test, since the testing will not progress erratically while time is taken to repair bugs. Clearly, the more care is taken during software development to produce bug-free code, the easier the testing that follows will be.

- **Observability**: Software tests examine the outputs produced by the software for particular inputs. This means that software is easier to test if the inputs produce distinct, predictable outputs. Further, it can be helpful to be able to examine the software's internal state, especially where there may be no predictable outputs, and especially when an error has been discovered.

- **Controllability**: As just mentioned, software testing involves examining outputs for given inputs. This means that the more easily we can provide inputs to the software, the more easily we can test the software. This implies that software is more testable when the tester has the ability to easily control the software in order to provide the test inputs. This controllability applies to the tests as well: tests should be easily specifiable, automated, and reproducible.

- **Decomposability**: When software can be decomposed into independent modules, these modules can be tested individually. When an error occurs in an individual module, the error is less likely to require changes to be made in other modules, or for the tester to even examine multiple modules.

- **Simplicity**: Clearly, the simpler the software, the fewer errors it will have, and the easier it will be to test. There are three forms of simplicity that software can demonstrate: **functional simplicity**, in which the software does no more than is needed of it; **structural simplicity**, in which the software is decomposed into small, simple units; and **code simplicity**, in which the coding standards used be the software team allows for the easy understanding of the code.

- **Stability**: If changes need to be made to the software, then testing becomes easier if these changes are always contained within independent modules (via, for instance, decomposability), meaning that the code that needs to be tested remains small.

- **Understandability**: Clearly, the more the testers understand the software, the easier it is to test. Much of this relates to good software design, but also to the engineering culture of the developers: communication between designers, developers and testers whenever changes occur in the software is important, as is the ability for the testers and developers to easily access good technical documentation related to the software (such as APIs for the libraries being used and the software itself).

Test cases and test case design

Test cases are controlled tests of a particular aspect of the software. The objective of a test case is to uncover a particular error. Testing software, then, is the development of a suite of such test cases, and then their application to the software.

Tests should be kept simple, testing for specific errors (or specific classes of errors) rather than testing whole branches of the software's functionality at a time. In this way, if a test fails the failure will point to a particular area of the software that is at fault.

Testing strategies

Just as it is important to develop software in a way that eases software testing, it is also important to both design tests well, and to have a good strategy as to how the software should be tested.

Testing is incremental: it begins by testing small, self-contained units and progresses to testing these units as they interact with each other. Ultimately, the software as a whole — with all units integrated — is tested.

Testing can be broken down into the following stages: **unit testing** (testing individual modules), **integration testing** (testing modules as they are brought together), **validation testing** (testing to see if the software meets its requirements), and **system testing** (testing to see how well the software integrates with the various systems and processes used by the customer).

Unit testing

Unit testing is concerned with testing the smallest modules of the software. For each module, the module's interface is examined to ensure that information properly flows to and from the module. The data structures which are internal to the module should be examined to ensure that they remain in a consistent state as the module is used. The *independent paths* (see the section called " Flow graphs, cyclomatic complexity and white-box testing " in this chapter) through the module should each be tested. Boundary conditions should also be closely examined to ensure that they are properly handled (such as, for example, not reading past the end of an array). Importantly, remember to test the error handling code and ensure that they handle and report errors correctly.

Unit tests can easily be incorporated into the development process itself: unit tests can be written while each module is written. Indeed, some software design methods (such as extreme programming, see Chapter 2, *Process and Model*) ask for unit tests to be developed before the software itself. Regression tests (a form of integration testing), too, can be incorporated into the development process — one way of doing so is to have them automatically run each night, after all code developed that day has been submitted to a central repository.

Integration testing

Once unit testing is complete, the next important step is to combine the separate modules and ensure that they function correctly together. One may think that because each module functions correctly by itself that they will function correctly together. This is not always the case. For example, module *A* might have certain expectations about the behaviour of module *B* that are not being met, causing *A* to function incorrectly.

It may seem that integration testing can be carried out by combining all modules at once and testing how they function. Unfortunately, this "big bang" approach can make it difficult to track an error down to any one particular module. A better approach is to combine modules together incrementally, testing their behaviour at every step. Each increment brings us closer to having the complete software, but each increment remains constrained enough for us to properly test.

There are two general methods for performing module integration: the *top-down* and *bottom-up* approaches. **Top-down integration** testing begins by creating the overall software where much of its functionality is provided by empty stub modules. These modules perform no function other than to allow the software to compile, and to provide a way to exercise the modules which are being tested. The stubs are then replaced with the actual modules, one at a time, beginning with the modules involved with user-interaction and ending with the modules which perform the software's functionality. As each module passes its tests, a new module is integrated.

Clearly, top-down integration is difficult because of the need to create stub modules. The proper testing of some modules may rely upon a wide range of behaviour from the sub-modules, and so either the testing must be delayed until the stubs have been replaced, or the stubs must offer much functionality themselves (and may themselves be buggy).

The alternative to the top-down approach is **bottom-up integration** testing: here, modules which do not rely on other modules are combined together to create the software. Because we begin with modules that do not rely on other modules, no stub code is needed at all. At each step we test the combined software. When all tests have passed we add another module together. Ultimately, all the modules will be combined into the functioning software.

Whether a top-down approach, a bottom-up approach, or a mixture of both approaches, is used, whenever a new module is added to the software in order to be tested, the *whole* software has changed. Modules which rely on the new modules may behave differently, and so once again *all*the modules have to be tested. **Regression testing** is the testing of the previously tested modules when a new module is added. Regression testing should occur at every incremental step of the integration testing process.

Validation testing

Unit and integration testing asks the question, "Are we developing the software correctly?" Validation testing, on the other hand, asks, "Are we developing the correct software?" In other words, validation testing is concerned with whether the software meets its requirements.

Validation testing focuses on the user-visible portions of the software, such as the user-visible inputs and outputs, and the software's actions. The tests examine these user-visible portions to ensure that they meet the software requirements. While not part of the software itself, the documentation should also be examined to ensure that they meet any requirements concerning them.

We have previously mentioned beta testing as the process of the customers themselves testing the software. This can be a useful tool in the validation testing process, since the developers cannot foresee exactly how the customers may use the software.

System testing

Software is always employed within some larger context, such as all the systems and processes which a business customer may have in place. **System testing** is concerned with how the software behaves as it integrates into the broader system in which it will be used.

For example, when the software fails or suffers from an error it must not cause the whole system that is using it to fail. **Recovery testing** examines how the software behaves when it fails. **Security testing** examines how well the software protects sensitive information and functionality from unauthorised access. **Stress testing** examines how the software functions under an abnormal load. While software may perform well by itself, its behaviour can be quite different when its used in a larger setting; **performance testing** examines the software's performance within the context of the system as a whole.

Testing advice

While the previous sections have mostly given advice and guidelines on designing the overall testing strategy, in this section we discuss more concrete advice on creating individual tests, with a focus on testing for implementation bugs (i.e., unit and integration testing) rather than validation and system testing.

An initial distinction to be made when creating a test is the difference between *white-box* and *black-box* testing. **Black-box testing** treats the module as an object whose inner-workings are unknowable, except for how the module handles its inputs and outputs. Black-box testing does not examine the module's inner state, and assumes that if the module correctly handles its inputs and outputs, then it is error free. **White-box testing**, on the other hand, also examines the module's inner state in an attempt to ensure that its internal operations are correctly performed, no matter how the module handles its inputs and outputs.

White-box testing allows us to conceivably test every line of code in the module: since we are examining the software's internal state, we can determine where and how that state changes, and so construct tests to exercise not only every line of code, but every logical choice that can be made when executing the code. This process of testing all lines of code and all logical choices in a software module is called **exhaustive testing**, and it is extremely important to realise that, except in the most trivial of cases, exhaustive testing is impractical to perform. To see this, one need only consider that whenever the software contains, for example, **if** statements within **if** statements, or loops within loops,

the number of logical paths through the software increases exponentially, and so does the time required to test each of these choices. Even software of only a few hundred lines of code can quickly require more time than is feasible to test every logical decision that could possibly be made by the software. How this exponential explosion of choice is to be handled is an important aspect of white-box testing.

Flow graphs, cyclomatic complexity and white-box testing

It will be useful to introduce some simple graphical notation for representing the execution flow of a program. While testing can (and often is) discussed without mention of flow graphs, they do provide a graphical tool for better describing the various testing processes.

Figure 9.1, "Flow graph notation", displays various examples from the notation. The nodes in the graph represent some unit of processing that must occur. All logical decisions which affect the flow of program execution are represented by the edges in a graph: when an **if** statement decides on which branch to take (its **then** or **else** branches), this is represented by multiple edges leading to separate nodes. Similarly, various loops, case statements, and so on, are represented in a similar way.

An **independent path** through a program is a path through a flow graph which covers at least one edge that no other path covers. Examine Figure 9.2, "An example flow-graph". The paths 1,2,3,5,7,8 and 1,2,3,5,7,5,7,5,7,8 are not independent, because neither of them have an edge which the other does not. However, the paths 1,2,3,5,7,8 and 1,2,4,6,8 are independent.

The set of all independent paths through a flow graph make up the **basis set** for the flow graph. If our testing can execute every path in the basis set, then we know that we have executed every statement in the program at least once, and that we have tested every condition as well.

Note

While we may have tested every line of code, and every condition, we have still not tested every possible combination of logical choices that could be made by the conditional statements during software execution. In this way we limit the explosion of test cases the exhaustive testing would produce. However, this does mean that *some* bugs may still escape detection.

The **cyclomatic complexity** of a flow graph informs us how many paths there are in the graph's basis set (in other words, how many independent paths there are needing to be tested). There are three ways in which it can be computed:

- By counting the number of regions in the flow graph.

- If E is the number of edges in the flow graph, and N the number of nodes, then the cyclomatic complexity is: $E - N + 2$.

- The cyclomatic complexity is also $P + 1$, where P is the number of nodes from which two or more edges exit. These are the nodes at which logical decisions controlling program flow are made.
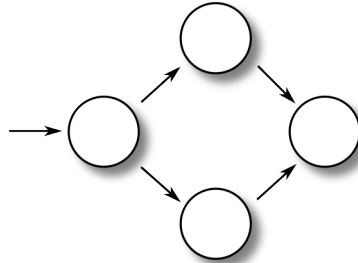
In Figure 9.2, "An example flow-graph", we can calculate the complexity in all three ways. There are four regions in the graph (remember to count the space surrounding the graph, and not only the spaces inside the graph). There are ten edges and eight nodes. There are three nodes from which two or more edges leave. Using the three methods above, we get:

- Four regions give a cyclomatic complexity of 4.

- Ten edges and eight nodes give a cyclomatic complexity of $10 - 8 + 2 = 4$

- Three nodes with two or more exiting edges gives a cyclomatic complexity of $3 + 1 = 4$
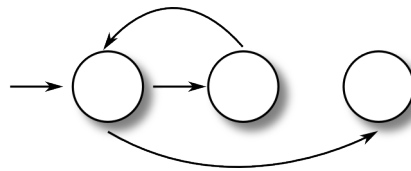
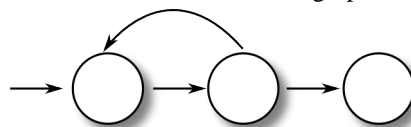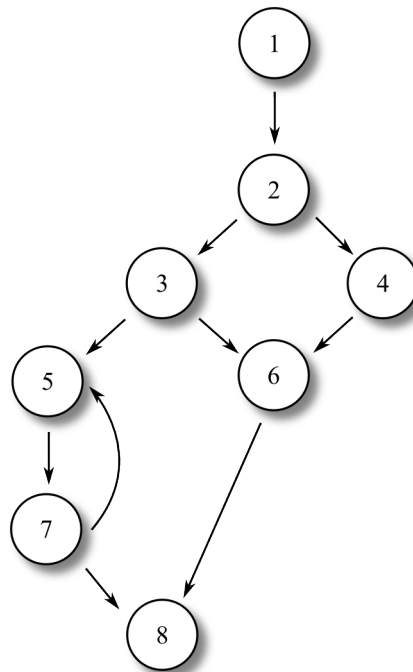# Figure 9.1. Flow graph notation
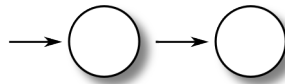
A sequence flow graph

An **if** statement flow graph

A **case** statement flow graph

An **until** statement flow graph

Figure 9.2. An example flow-graph

An example flow graph, with labeled nodes. Execution begins at node 1, and proceeds through an **if** statement, and possibly through a loop. Execution terminates at node 8.

As we can see, each method agrees with each other, and states that there are four independent paths through the program.

The third definition has broad applicability, since it provides a method for calculating complexity without using flow graphs: one can merely count the number of conditional statements in a program and add one to this number.

These concepts of independent paths, basis sets and cyclomatic complexity are important to testing, because they give us a concept of how well our tests may be exercising the code. Importantly, those portions of the code which are used least are the portions which are the least likely to be tested, and the most likely to retain their errors. Discovering the independent paths through a program, and testing them all, allows us to ensure that these errors do not go unchecked. Various studies have also shown that the higher the cyclomatic complexity of a given package, the more likely it is to have errors.

We want to again point out that testing all independent paths is not the same as exhaustive testing. Exhaustive testing wishes to test *all* possible paths through the flow graph, as determined by examining all possible combinations of logical choices that could occur at the conditions. In the particular example used here, the number of all paths through the program depends on the number of times that the loop needs to iterate. If the loop were to contain other loops, or **if** statements, this number of paths would increase dramatically. The use of independent paths keeps the number of tests to a reasonable level, while ensuring that all lines of code are tested.

The testing methodology which tests all independent paths through an application is called **basis path testing**, and is clearly a white-box testing method.

Basis path testing can be tedious to perform. It does provide suggestions for determining tests in general, however. When determining how to test code, always test the logical conditions (this is called **condition testing**). Also, focus on the conditions and validity of loop constructs (**loop testing**).

Black-box testing

While white-box testing is interested in *how* the module performs its function, black-box testing is interested only in *what* the module should be doing. Black-box testing tests the *requirements* of the software module, and not at all with how it manages to meet these requirements.

These requirements cover a wide range of areas to be tested, and includes:

- Input and output errors, which includes not only errors which may occur when making use of the software module, but also errors that may occur when the software module attempts to use other modules, such as a database.

- Incorrect and missing functions.

- Initialisation and termination errors.

- Behaviour errors.

- Performance errors.

- Reliability errors.

We will examine two methods of black-box testing: *equivalence partitioning* and *boundary value analysis*.

Equivalence partitioning

Equivalence partitioning divides a software module's input data into equivalence classes (note that these are not classes in the sense of object-oriented programming). The test cases are then designed so as to test each one of these input classes; a good test case will potentially discover errors in whole classes of input data.

Generally, for any input, we will have at least *two* classes. For example, if the input to the software is a Boolean, then this is clearly the case (in this case, each class has one value: one true, the other false). Similarly, if the input is a member of a set, then you will have multiple classes. For example, if we had a software module from a graphics package which, when given a rectangle, used the lengths of its sides to determine whether the rectangle was a square or not, then we would have two classes: one for the rectangles which are square, one for rectangles which are not.

The number of classes strongly depends on the type of the input data. If the input data requires a specific number, then there are *three* classes: one for that number, one for all numbers less than it, and one for all numbers greater than it. Similarly, if the input should be from a range of numbers, we again have three classes.

Testing each input class reveals whether the software module correctly handles the range of input that it could receive.

Boundary value analysis

Software bugs tend to occur more frequently at their "boundary values", which are those values around which a conditional changes the value it evaluates to. For instance, boundary values are those values for which an **if** statement will change between choosing to execute its **then** or **else** portions, or where a loop decides whether to iterate or not.

This increase in errors can occur for simple reasons, such as using a greater-than comparison instead of a greater-than-or-equal-to comparison. When looping, common boundary mistakes include iterating one time too many, or one time too few.

Because of the increased frequency with which errors occur around boundary values, it is important to design test cases that properly exercise the boundaries of each conditional statement. These boundaries

will occur between the various input classes in the equivalence partitioning method, and so boundary value analysis is well suited to being combined with that method.

Object-oriented testing

Testing methodologies can be modified slightly when the software is developed in an object-oriented manner.

The basic "unit" for unit testing becomes the class as a whole. This does have the consequence, however, that the various methods cannot be tested in isolation, but must be tested together. At the very least the class's constructor and destructor will always be tested with any given method.

Similarly, when performing integration testing, the class becomes the basic module which makes up the software. **Use-based** is a bottom-up integration method constructing the software from those classes which use no other, then integrating these with the classes which use them in turn, and so on. Classes can also be integrated by following program execution, integrating those classes that are required in order to respond to particular input (**thread-based** testing), or, similarly, to integrate those classes which provide some specific functionality, independent of the input to the software (**cluster** testing).

In general, you should not only test base-classes, but *all* derived classes as well. Inheritance is a special case of module integration, and should be treated as such.

Debugging

Once a test case has been executed and a bug located, debugging begins. **Debugging** is the process of locating the cause of a software error and correcting it. Unfortunately, this is not necessarily an easy process. Software engineers are only ever presented with the software's *behaviour*, and they do not directly see the error's *cause*. Debugging, then, relies on the software engineer to determine from the software's incorrect behaviour the causes of this behaviour.

Debugging begins with the software failing a test case. The software engineer debugging the software will then hypothesise a possible cause and implement the needed changes to correct this in the software. The failed test is then rerun. Further test cases may also be written to help narrow down the actual cause. This all occurs iteratively, with each step hopefully providing more information to the developer as to the root cause of the error.

A number of debugging tactics have been proposed. They can be used alone, although they become far more effective when used in combination with each other.

Brute force debugging

This is conceptually the simplest of the methods, and often the least successful. This involves the developer manually searching through stack-traces, memory-dumps, log files, and so on, for traces of the error. Extra output statements, in addition to break points, are often added to the code in order to examine what the software is doing at every step.

Backtracking

This method has the developer begin with the code that immediately produces the observable error. The developer than backtracks through the execution path, looking for the cause. Unfortunately, the number of execution paths which lead to any given point in the software can become quite large the further the cause of the bug is from where the error occurs, and so this method can become impractical.

Cause elimination

In this method, the developer develops hypotheses as to why the bug has occurred. The code can either be directly examined for the bug, or data to test the hypothesis can be constructed. This method can

often result in the shortest debug times, although it does rely on the developers understanding the software well.

Bisect

Bisect is a useful method for locating bugs which are new to the software. Previous versions of the software are examined until a version which does not have the error is located. The difference between that version's source code and the next is then examined to find the bug.

Review

Questions

Review Question 1

Complete:

Testing is the [_____] process of running software in with the intent of [_____] in the software. It is important to realise that testing is a [_____] process, and is not the accidental discovery of software bugs.

A discussion of this question can be found at the end of this chapter.

Review Question 2

Who are the different parties involved in software testing, and how does the testing shift from one party to another?

A discussion of this question can be found at the end of this chapter.

Review Question 3

What guidelines would you give for developing software that is easily testable?

A discussion of this question can be found at the end of this chapter.

Review Question 4

Complete:

Test cases are controlled tests of [_____] of the software. The objective of a test case is [_____]. Testing software, then, is the development of [_____], and then their application to the software.

A discussion of this question can be found at the end of this chapter.

Review Question 5

What are the different stages of software testing?

A discussion of this question can be found at the end of this chapter.

Review Question 6

What is the difference between white-box and black-box testing?

A discussion of this question can be found at the end of this chapter.

Review Question 7

Why is it not practically possible to test every logical path through a piece of software? What alternatives are there?

A discussion of this question can be found at the end of this chapter.

Review Question 8

Debugging begins with the software [_____] a test case. The software engineer debugging the software will then [_____] a possible cause and [_____] in the software. The failed test is then rerun. This might not always provide an exact reason for the bug, and so further test cases may also be written to [_____]. This all occurs iteratively, with each step hopefully providing more information to the developer as to the root cause of the error.

A discussion of this question can be found at the end of this chapter.

Review Question 9

What are some common debugging techniques?

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Testing is the **planned** process of running software in with the intent of **discovering errors** in the software. It is important to realise that testing is a **purposeful** process, and is not the accidental discovery of software bugs.

Discussion of Review Question 2

There are three parties involved in software testing:

1. The developers

2. Independent testers

3. The customers / users

Testing originally begins with the *developers*, who need to ensure that the software they have coded works as they intend. Testing is then taken over by *independent testers*, who are able to use more specialised testing frameworks than the developers themselves. They can also perform user-based testing, which most developers would not be able to do.

Finally, as the software is delivered to the customer and the software begins to be used, its users become the final group of testers.

Discussion of Review Question 3

- *Operability*. New code developed for relatively bug-free software will have fewer bugs than code developed for relatively buggy software.

- *Observability*. Software whose internal state can easily be examined, and that produces well defined outputs for particular inputs, is easier to test.

- *Controllability*. If the tester has the ability to easily control the software, testing becomes easier: controllability allows for the easier inputting of data and examining of output.

- *Decomposability*. Independent modules can more easily be tested, and changes made to a module are less likely to affect other modules.

- *Simplicity*. The simpler the software, the fewer errors it will have.

- *Stability*. Testing is easier if changes made to correct bugs are limited to independent modules.

- *Understandability*. The better the testers understand the software — through good software design, documentation, and so on — the better they can test the software.

Discussion of Review Question 4

Test cases are controlled tests of **particular aspects** of the software. The objective of a test case is **uncover a particular error**. Testing software, then, is the development of **a collection of test cases**, and then their application to the software.

Discussion of Review Question 5

- *Unit testing*, which tests the smallest modules of the software.

- *Integration testing*, which tests the software as the modules are brought together.

- *Validation testing*, which tests the software to ensure that it meets its requirements specification.

- *System testing*, which examines how well the software integrates with the various systems and processes used by the customer.

Discussion of Review Question 6

Black-box testing tests the module as an object whose inner-workings are unknowable, except for how it handles its inputs and outputs. Black-box testing therefor does not examine a module's inner state, only the interfaces to the module. It assumes that if it handles its inputs and outputs correctly, then the module itself behaves correctly.

White-box testing examines the software's internal state; it does not assume that because a module has handles its inputs and outputs correctly that the module has behaved correctly. For example, after correctly outputting some data, a module may leave its internal state with invalid values, and so any further action will fail.

Discussion of Review Question 7

The number of possible logical paths through a piece of software can grow exponentially as *if* statements and loops are added. This would make the amount of time required to test a large software package prohibitive.

An alternative testing strategy is to determine the software's cyclomatic complexity, which tells us how many independent paths there are through the software. Each of these independent paths can then be tested; this will provide good code coverage (it will execute every line of code) without exhaustively testing every logical path through the software, and so greatly reduces the amount of time needed to execute the software.

Discussion of Review Question 8

Debugging begins with the software **failing** a test case. The software engineer debugging the software will then **hypothesise** a possible cause and **implement the needed changes** in the software. The failed test is then rerun. This might not always provide an exact reason for the bug, and so further test cases may also be written to **narrow down the cause**. This all occurs iteratively, with each step hopefully providing more information to the developer as to the root cause of the error.

Discussion of Review Question 9

- *Brute force debugging*; the developer searches through stack-traces, memory-dumps, log files, and other data generated by the program to locate the error.

- *Backtracking*; the developer examines the code that immediately produces the observable bug, and then moves backwards through the execution path until the cause of the bug has been found.

- *Cause elimination*; the developer hypothesises reasons why the bug has occurred, and then either directly examines the code to see if these reasons exist, or produces further tests to narrow down the choice between various hypotheses.

- *Bisect*; the developer examines previous versions of the software until one without the bug is located. The difference between that version of the source code and the next (in which the bug does not exist) is where the bug will be located.