Heinemann

Software Design and Development

HSC Course

Allan Fowler

Heinemann

Heinemann
A division of Reed International Books Australia Pty Ltd
22 Salmon Street, Port Melbourne, Victoria 3207
World Wide Web hi.com.au
Email info@hi.com.au

Offices in Sydney, Brisbane, Adelaide and Perth.
Associated companies, branches and representatives throughout the world.
© Allan Fowler 2000
First published 2000
2003 2002 2001 2000
10 9 8 7 6 5 4 3 2 1 0

Publisher: Rosie Adams
Editor: Felicity Shea
Designer: Gerry Theoharis
Design development: Giulia De Vincentis
Cover designer: Relish Design
Illustrations: Idczak Enterprises
Photograph researcher: Gwenda McGough

Typeset in 10/11.5 Berling by Idczak Enterprises
Film supplied by Typescan, Adelaide
Printed in Hong Kong by H&Y Printing

National Library of Australia
cataloguing-in-publication data:

Fowler, Allan, 1949–
Heinemann software design and development:
HSC course.

Includes index.
ISBN 0 86462 513 8 (HSC course).

1. Computer science. 2. Computer software—Development.
I. Title

005.3

Contents

Introduction

This book has been written to support students and teachers in implementing the NSW Software Design and Development HSC Course. The course has been introduced by the Board of Studies to give students who are interested in the field of software design and development an opportunity to study the subject and create appropriate software solutions.

The development of software involves careful planning, clear documentation and an appreciation of the effects that the product may have on members of society. To this end, the book covers aspects of development, including analysing the problem, planning a solution, creating and testing the solution, documenting the solution as well as the associated social and ethical considerations.

Since different problems will need different approaches, a number of methods of software development are examined. These vary from the formal, structured approach through to the informal approaches such as end-user development. The nature of the problem to be solved will also dictate which computer language is most suitable to develop the solution. To cater for this need, examples have been drawn from a wide range of languages.

Each chapter begins with a statement of the chapter outcomes, student knowledge and student experiences. This allows both the teacher and the student to ensure that the syllabus content has been met. The chapters end with a summary, a set of review exercises and a team project.

Teamwork is an important part of software development as many projects are too large and complex for a single individual to tackle. The team projects have been designed to enable students to experience working members of a team. Each of the team projects covers one or more of the aspects of software design and development that have been covered in the chapter. Thus, the projects will also give students the opportunity to review and discuss the material presented in the chapter.

Additional support for teachers and students using *Heinemann Software Design and Development* is available on the Internet.

Visit http:/www.hi.com.au/softwaredesign.

About the authors

Allan Fowler has taught Computing Studies to senior students at Gosford High School since the beginning of 1978. The first courses were school-based programming courses as Other Approved Studies. He then taught the 2 Unit course, first examined in 1990, and was a participant in the support-material writing workshop held in Port Macquarie that year. He has contributed computing related articles to professional journals in both computing and mathematics and has given many workshops and presentations to teachers, parents and students in computing.

Allan wrote the very successful *Heinemann Senior Computing Studies 3 Unit (Additional) HSC* and contributed to the *Heinemann Senior Computing Studies 2/3 Unit Common HSC Course*. He wrote the solution manuals for both texts. . He is the author of the companion volume to this book—*Heinemann Software Design and Development: Preliminary Course.*

Dieter Opfer is a Computer Co-ordinator and Network Administrator at Sefton High School, and is also the Vice-President of the NSW Computer Education Group. He is currently undertaking a PhD investigating the use of the Internet in Distance Education.

Errol Chopping is a Lecturer in Computer Science at Charles Sturt University—Mitchell Campus, specialising in Software Development.

Paul Jenner is Head Teacher at Kanahooka High School. He has been active in the development of computer education for many years, having held several positions as a state and regional Computer and Technology Education Consultant.

Rick Walker is Head of Information Technology at Hunter Valley Grammar School and has taught Computer Studies for 15 years.

Acknowledgments

I am indebted to my wife Kaye and my children Katharine and Stephanie for their patience and understanding throughout the writing of this text. I also thank my parents Hilda and Roy for the encouragement they have given me throughout my life.

Comments and suggestions made by Cathy Webber, Anthony Connolly and Dieter Opfer have significantly assisted me in the task of producing this text. I would particularly like to thank the authors who contributed to this text, and Glenda Horner for her criticisms and suggestions for improvements to the early drafts.

Finally I would like to thank Rosie Adams and the team at Heinemann for the very professional help and encouragement that they have willingly given.

Allan Fowler

Heinemann Software Design and Development and the HSC Course Outcomes

The grid shows how the chapters in *Heinemann Software Design and Development: HSC Course* link with the HSC Course Outcomes.

HSC Course Outcomes A student needs to:		Chapter(s)
H 1.1	explains the interrelationship between hardware and software	5, 8, 9, 10
H 1.2	differentiates between various methods used to construct software solutions	2, 3, 5, 7, 8, 9
H 1.3	describes how the major components of a computer system store and manipulate data	5, 7, 8, 10
H 2.1	describes the historical developments of different language types	9
H 2.2	explains the relationship between emerging technologies and software development	1, 2, 5, 9
H 3.1	identifies and evaluates legal, social and ethical issues in a number of contexts	1, 2, 3, 5, 6, 7, 8, 9, 10
H 3.2	constructs software solutions that address legal, social and ethical issues	3, 5, 6, 7, 8, 10
H 4.1	identifies needs to which software solutions are appropriate	3, 4, 8, 9, 10
H 4.2	applies appropriate development methods to solve software problems	2, 3, 4, 5, 6, 7, 8, 9
H 4.3	applies a modular approach to implement well-structured software solutions and evaluates their effectiveness	3, 4, 5, 6, 7, 8
H 5.1	applies project management techniques to maximise the productivity of the software development	2, 3, 5, 6, 7, 8
H 5.2	creates and justifies the need for the various types of documentation required for a software solution	2, 3, 5, 6, 7, 8
H 5.3	selects and applies appropriate software to facilitate the design and development of software solutions	2, 3, 5, 6, 7, 8
H 6.1	assesses the relationship between the roles of people involved in the software development cycle	2, 3, 6, 7, 8
H 6.2	communicates the processes involved in a software solution to an inexperienced user	2, 3, 5, 6, 7, 8
H 6.3	uses a collaborative approach during the software development cycle	3, 5, 6, 7, 8
H 6.4	develops effective user interfaces, in consultation with appropriate people	3, 6, 7, 8

chapter 1

Social and ethical issues

Outcomes

A student:

- explains the relationship between emerging technologies and software development (H 2.2)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)

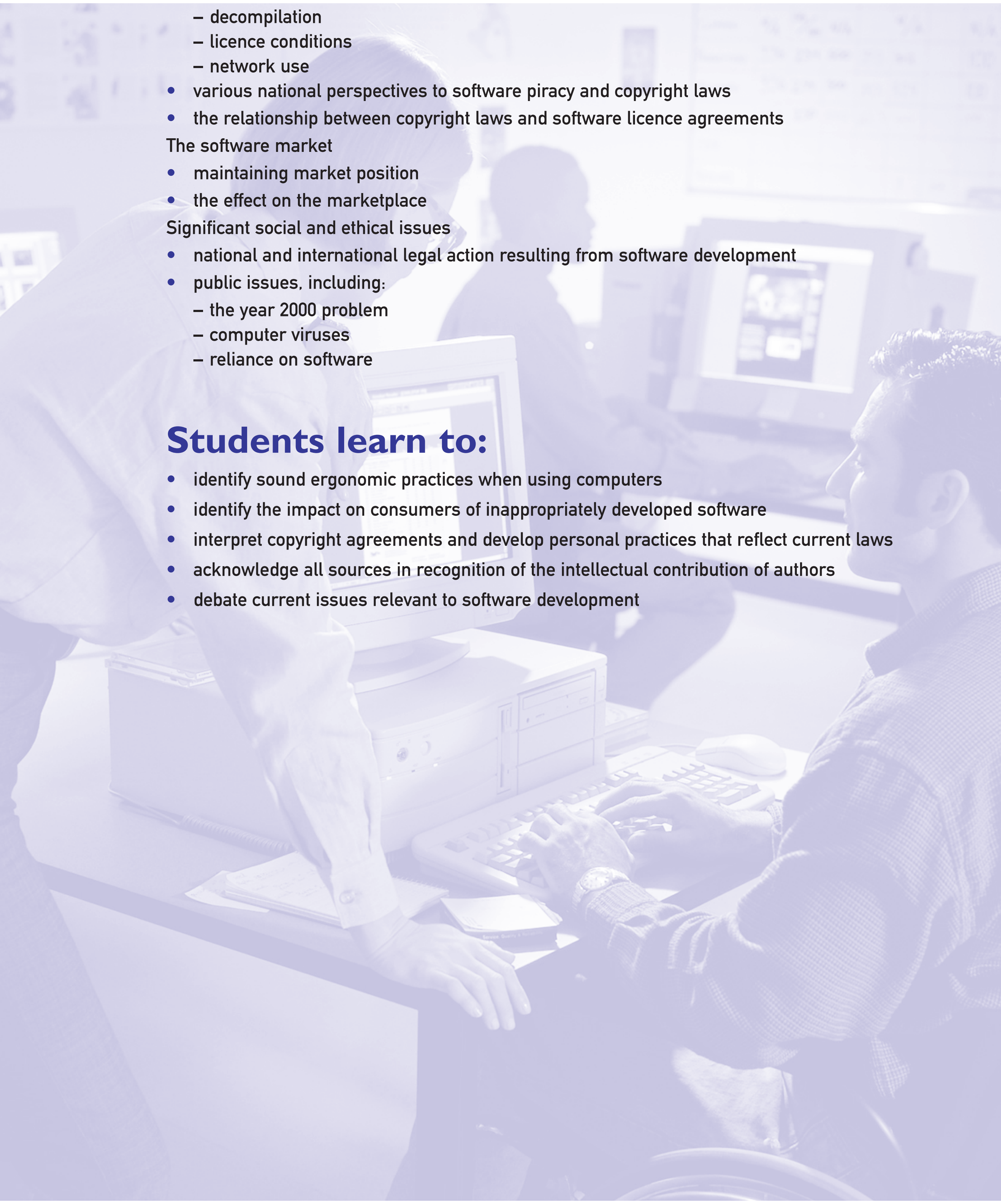
Students learn about:

Rights and responsibilities of software developers

- authorship
- reliability
- quality
- response to problems
- code of conduct
- viruses

Software piracy and copyright

- concepts associated with piracy and copyright, including:
  – intellectual property
  – plagiarism
  – shareware
  – public domain
  – ownership versus licensing
  – copyright laws
  – reverse/backwards engineering



  - decompilation
  - licence conditions
  - network use
- various national perspectives to software piracy and copyright laws
- the relationship between copyright laws and software licence agreements

The software market
- maintaining market position
- the effect on the marketplace

Significant social and ethical issues
- national and international legal action resulting from software development
- public issues, including:
  - the year 2000 problem
  - computer viruses
  - reliance on software

Students learn to:

- identify sound ergonomic practices when using computers
- identify the impact on consumers of inappropriately developed software
- interpret copyright agreements and develop personal practices that reflect current laws
- acknowledge all sources in recognition of the intellectual contribution of authors
- debate current issues relevant to software development

Preliminary review

In the Preliminary Course it was noted that the results of a creative process could be classified as intellectual property. Copyright allows the creator of such items to be acknowledged and compensated for the time and effort involved. Software is the result of such a creative process and is therefore subject to the laws of copyright.

The acquisition of software, whether it is purchased, freeware or custom developed, brings with it a responsibility to ensure that users adhere to the conditions of any licence. Licences generally give the purchaser the right to use the software, but they do not give the purchaser ownership of the code. Since the purchaser does not own the code it may not be used in any way other than that specified in the licence. This means that the code cannot be modified or backward engineered.

A software licence is a legal document. By accepting the conditions of a licence, an individual or organisation has entered into a contract. The licence contract specifies the number of computers that can be installed with the software as well as the way in which backups can be managed. The licence agreement is binding to the same extent whether the software has been acquired commercially, is shareware or is in the public domain (freeware).

Rights and responsibilities of software developers

Development of software solutions carries with it a number of rights and responsibilities. Software developers have the right to be acknowledged as the author of an application in the same way that a composer has the right to acknowledgment for a tune. Software authors also have responsibilities towards the users of their software. These responsibilities include:

- claiming authorship only of what they have written themselves
- creating reliable software
- creating quality software
- responding to problems
- following an ethical code of conduct
- preventing the spread of viruses.

Authorship

As noted, the creative process brings with it the right of an author to be acknowledged for a work. The developer has two responsibilities: the first is to protect his or her interests in the system being developed; the second is to protect the rights of other authors whose modules are incorporated into the final solution.

When a software solution is created, the developer needs to ensure that the sources of any previously written code are acknowledged. It is the developer's responsibility to clear the use of that code with the copyright holder. If for any reason this clearance cannot take place, the module cannot be used and one must be developed from the start. Even then, care needs to be taken in the development process to ensure that the module developed is not plagiarised from the one for which permission has not been given. At times this is very difficult and many legal battles have been fought over the similarity of software titles.

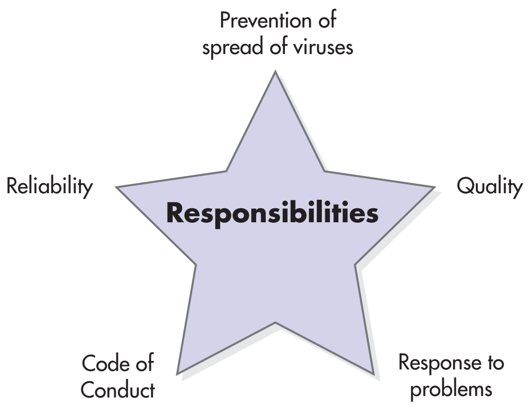


Figure 1.1 A software developer has a number of responsibilities.

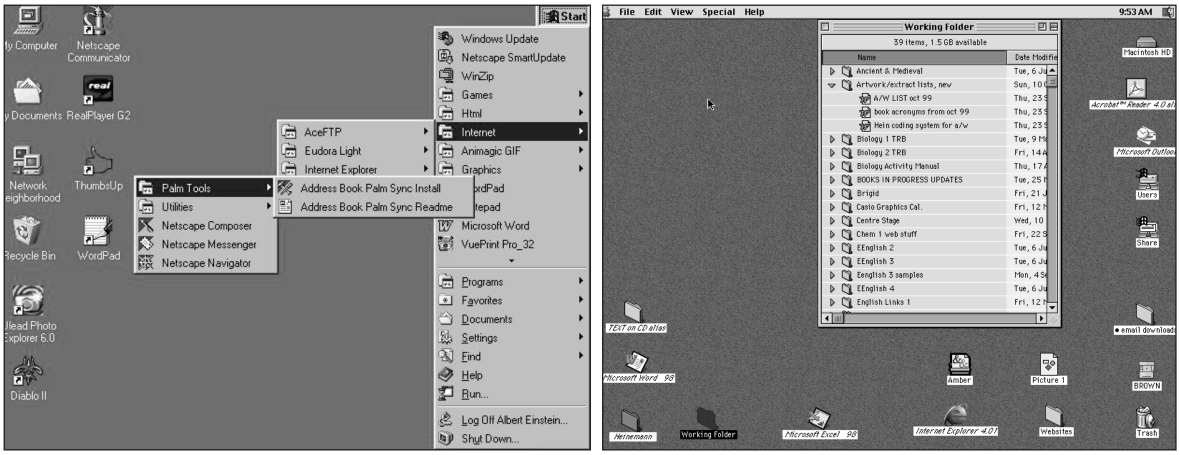


Figure 1.2 The Windows operating system (left) and the Macintosh operating system (right) look similar in many respects. This has been the cause of a long court battle in the United States.

Reliability

A software developer has a responsibility to make sure that the program being developed is able to perform its task effectively and reliably.

Reliability refers to the ability of the software to perform without failure. Software failure occurs when the software performs in an unpredictable way.

A software failure may occur as a result of a fault in the software. However, faults are not the only possible source of failure. Anything that causes an unpredictable result in the software contributes to failure. For example, the introduction of a new operating system or hardware driver may cause a conflict with a program.

Figure 1.3 A single failure of software may have a disastrous effect.

In some cases software failure in itself may not greatly affect the reliability of a program. However, in other cases a failure will have a great effect on the reliability. It cannot safely be stated that if a failure occurs in a rare set of circumstances it will not greatly affect the reliability of the software. The rare failure may have a disastrous effect on the system, whereas a frequent failure may be predictable, have a minimal effect on the system and produce a result that is easily corrected. For example, a fault in a program may, when a file reaches a certain size, overwrite other files on a hard drive. Just one occurrence of this fault is sufficient to cause concern, as it produces a disastrous result. If a program sorted lists in the wrong way (for example, if a descending sort was requested and it sorted in an ascending order and vice versa), this fault could be easily overcome.

The issue of reliability is a complex one. The software author can improve the reliability of the coded solution by including code to thoroughly check input data and operational aspects of the code, such as the size of files produced. However, this approach does have a couple of disadvantages as it will decrease the execution speed of the program and will also increase the cost of development. One of the ways in which software developers can increase the reliability of their products is by keeping to a well-defined and standard process for software development.

Quality

The quality of software is measured by a number of criteria. These criteria include factors such as reliability, ease of use, consistency of the user interface, documentation and the adherence to software standards. Since software is a product, the principles of quality assurance (QA) are often applied to ensure the quality of the product.

The importance of *reliability* and how it may be increased has already been discussed.

Ease of use is a general term used to describe the comfort of the user with the software. Easy-to-use software will exhibit a consistent interface that is intuitive to the user. The interface will often mimic a non-computer system that is familiar to the user. A program that is easy to use is, by implication, easy to learn.

A consistent user interface is predictable from screen to screen. As discussed in the Preliminary Course, *consistency* involves the placement of similar items in the same place from screen to screen. Consistency also involves the use of the same screen element in each of the screens to represent the same task. Using the same typeface for the same or similar menu items also contributes to consistent screen design.

The quality of the *documentation* produced during the development cycle contributes to the overall quality of the product. It is important that both the process documentation created as part of the development cycle and the product documentation created for the end user are of high quality. Process documentation has to be used during any maintenance of the software and should therefore be written to a predetermined standard. Product documentation is often the only source of information for a user and so it must be written in a manner that encourages the user to refer to it.

Software *standards* have been created by many organisations. The standards cover aspects such as the form of documentation, algorithm description, coding and the user interface. By using a set of predetermined standards, it is easier for a developer to produce modules that fit in with the overall product. The strict adherence to standards will also help in the development stage as it assists new members of a team to quickly understand their role in the development process.

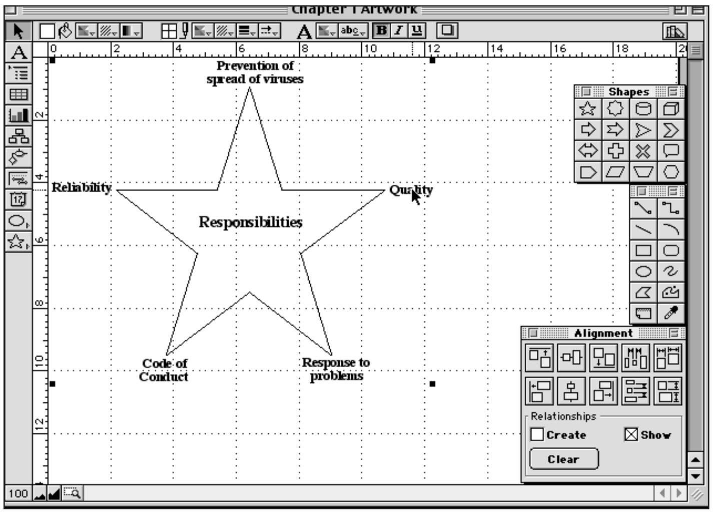


Figure 1.4 An easy-to-use program will have an intuitive interface.

Response to problems

In all but the simplest of programs a developer will find that all does not proceed simply. Programmers have a continual battle with problems of varying sizes. For many programmers this is the most exciting aspect of the job. However, whether excited or frustrated, the programmer's task is to overcome the problem. If the program is to succeed, the problem must be overcome in an orderly fashion.

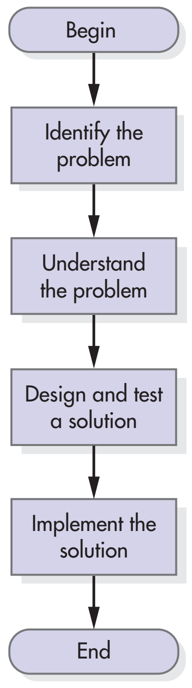


Figure 1.5 When development problems occur, the normal problem-solving process is used.

It is tempting to identify the solution and proceed to solve it without regard to the formal steps of problem solving. Taking this path can be dangerous or, at least, inefficient. By proper documentation of the solution the programmer can analyse what happened and either avoid the problem or anticipate a similar problem and manage the solution in the future. The formal steps are virtually the same as the steps taken in the development of software:

- Identify the development problem.
- Understand the development problem.
- Design and test a solution to the development problem.
- Implement the solution to the development problem.

Code of conduct

As already seen, there are a number of ethical and social issues that confront a software developer. It is the responsibility of the developer to ensure that all these issues are addressed throughout the development process. Some of the areas, such as the respect for copyright and the abidance to software licences, are legal requirements, whereas others rely on the ethics of the developer. Each software author should uphold the written laws and rules as well as exhibiting a high standard of ethics.

The ethical behaviour required of a developer should at least cover the following areas:

- acknowledgment of the contributions made by all those who have assisted in the development process
- clearance to use copyrighted modules or those written by others
- privacy of personal details, security of data and the maintenance of business secrets
- the use of appropriate prompts within the program so that it is non-threatening, non-racial and inclusive.

In many software development agencies, a company code of conduct is maintained. Where this is the case, the employee may be held accountable for any breaches of the code.

Viruses

Viruses pose a serious threat to many computer systems. They can be introduced by a number of different means. The software developer has a great responsibility to ensure that viruses are not spread by means of the development process. This means that the developer needs to have the capability to detect and eradicate the most recent viruses as well as procedures to ensure that they do not receive or transmit them.

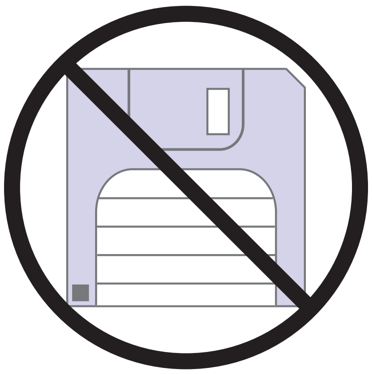


Figure 1.6 By not using pirated software, developers respect the rights of others as well as preventing the spread of viruses.

Precautions that should be taken to avoid the problem of viruses include:

- scanning of any removable media for viruses before use (removable media include floppy disks, disk cartridges and storage devices such as external hard disks.)
- careful vetting of email so that malicious emails are identified and removed before they cause problems
- not mixing home and work applications on the same computer (for example not using work computers for game playing or Internet surfing)
- keeping up to date with the latest viruses by updating the definitions for the virus software regularly.

Exercise 1.1

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

A software __________ has a right to be acknowledged as the __________ of an application only if it has been __________ by __________. This right of the author is __________ by the laws of __________. Software is an example of an __________ property in the same way that a __________ is. __________ property is the result of a __________ process.

2 List and briefly describe the rights and responsibilities of a software developer.

3 Create a checklist of criteria that you think are important in quality software. Save this checklist for use when you develop your project.

4 Discuss the ethics of the student involved in the following fictitious case.

A student researches an assignment on the Internet and happens to find a school site where another student has posted an essay that exactly answers the question. The student downloads the essay and passes it off as an original work.

5 Investigate a number of standards that may apply to the documentation of a software application. Create a document that outlines the standards that you will follow when you write the documentation for your software project.

Software piracy and copyright

As discussed in the Preliminary Course, the author of a software title has the right to be acknowledged and rewarded for the thought and effort expended during its creation. The laws of copyright are designed to protect that right. The illegal copying of software, commonly known as piracy, violates these laws and also deprives the copyright owners of their financial reward.

Concepts associated with piracy and copyright

A number of terms are used when discussing the twin issues of copyright and piracy.

Intellectual property

Intellectual property is a work resulting from some creative activity. The creative process automatically assigns ownership to the creator. If the object is not presented in a physical form, for example a tune or a poem, its ownership is harder to establish. The concept of intellectual property is designed to give a creator ownership of the result of that creative process. The term 'intellectual property' really states that the created object is the result of a thought (or intellectual) process.

Plagiarism

Plagiarism refers to the process of claiming authorship of an intellectual property owned by another person. For example, if you download an article from the Internet, hand it in as an assignment and claim that you have written it, you would be guilty of plagiarism. An article is considered to be plagiarised if it is substantially the same as one that already exists.

Shareware

Shareware is software that is distributed free of charge, but after a trial period of use the user is expected to make a payment to the author. The reasoning behind shareware is that it allows small developers to create a software title and distribute it without the need to use commercial distribution channels such as stores. By creating a direct link between the author and the user, the cost of shareware is usually a lot less than that of an equivalent commercial title. Shareware can usually be copied and distributed by anyone as long as all the electronic documentation and copyright notices are also distributed.

Public domain

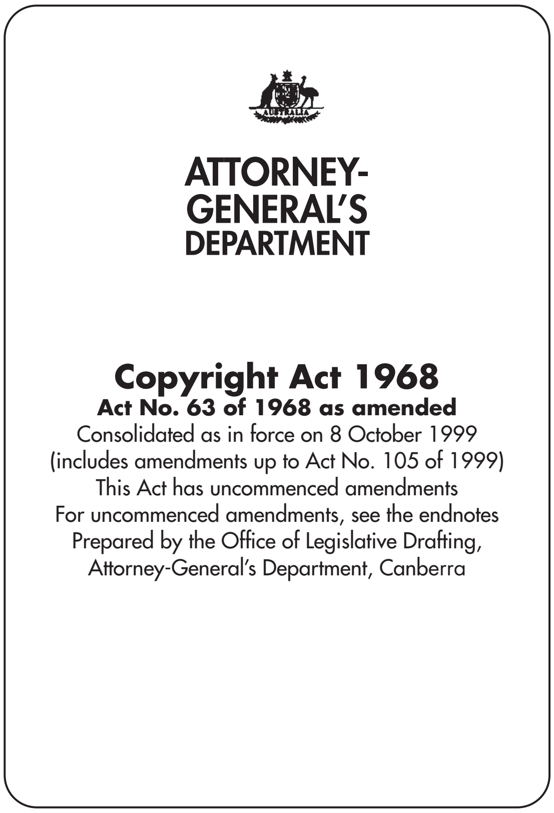
Public domain software is also freely distributable. However, unlike shareware, the author is not paid for the right of an individual to use the software. Even though there is no payment for public domain software, the copyright still remains with the author. Some public domain software is freely distributable for individual use, but may need to be licensed in the usual way if used in a commercial situation. The licence agreement that accompanies a public domain title will list the conditions under which the licence is granted.

Ownership versus licensing

A common misconception among computer users is that when they purchase software they own it. This is not the case: a licence accompanies software. The terms of the licence grant the purchaser the right to use the software but not ownership of it. The only time that ownership of software will be purchased is when customised software is written for an organisation. In this case the purchaser then owns the software. Even in the case of customised software, there may be parts of the application that are licensed from an external source. For example, a database may be created from an application program such as Access or Filemaker. In this case the application program will be licensed in the normal way, with the file or files created for the solution being owned by the purchaser.

Copyright laws

Copyright laws protect the rights of an author. The *Copyright Act 1968* (Cwlth) (Act No. 63 of 1968 as amended) governs copyright in Australia. The Act has been amended on a number of occasions to include new technologies such as computer programs. The Act defines what is subject to copyright, the length of time that copyright in a work exists, what is and is not an infringement of copyright, the duties and responsibilities of the Copyright Tribunal and the penalties that can be imposed for breaches of copyright. The full text of the Copyright Law is available to download from the Internet (at the time of writing, the Act was available from the federal parliament by using the search site at http://scaleplus.law.gov.au/home/ mainpage.html). As copyright is a very complex field, the law is a very large document. A majority of other countries also have strong copyright laws. In most cases, the granting of a copyright in one country means that the copyright automatically applies in other countries.



ATTORNEY-GENERAL'S DEPARTMENT

Copyright Act 1968
Act No. 63 of 1968 as amended
Consolidated as in force on 8 October 1999
(includes amendments up to Act No. 105 of 1999)
This Act has uncommenced amendments
For uncommenced amendments, see the endnotes
Prepared by the Office of Legislative Drafting,
Attorney-General's Department, Canberra

Figure 1.7 Copyright in Australia is governed by the *Copyright Act 1968*.

Reverse/backward engineering
The process of reverse (or backward) engineering involves examining an existing system to understand its components and using that understanding to create a similar system. For example, if one car company develops a new type of engine, a second company may buy one or more cars containing the new engine and use reverse engineering to develop their own version.

Decompilation
Decompilation is the process of translating executable machine code into assembler language so that the structure of the program can be more easily understood. Decompilation may be useful when the original coding of a solution has been lost; however, its use to reverse engineer a copyrighted application is definitely a breach of both the licence agreement and the laws of copyright.

Licence conditions
The conditions of a software licence spell out exactly how the software may be used by the licence purchaser. The licence will state the number of computers that can have the software installed on them, what is meant by terms such as 'backup copies' and, in the case of shareware and public domain software, the rules for free distribution. The licence is a legally binding contract and it is up to the purchaser to ensure that this contract is obeyed.

Network use
The licence will state how the software may be used on a network. The term network refers to a number of connected computers. A network licence may be purchased for a particular number of users or it may apply to a site. Licensing of network software also includes the network and computer operating systems. It is up to the network administrators to ensure that all network software is properly licensed. When updating network software, again administrators must ensure that sufficient licences have been purchased to cover the use of the software.

Software piracy and copyright laws
The *Copyright Act 1968* is a federal law that grants creators their rights at a national level. This ensures that unscrupulous people do not exploit differences in state copyright legislation. The federal legislation also makes provision for an Australian copyright to be recognised internationally. The legislation also describes the composition and function of the Copyright Tribunal.

As well as protecting the individual rights of a creator, the Copyright Act also helps to ensure that the publishing industries within Australia are financially viable. Development of creative works, including computer software, is an expensive process, and unless appropriate rewards are available, investment in this process will not be made. The long-term effect of this on the Australian economy would be disastrous, as those Australians with the creative skills and temperament would leave the country, forcing us to import what we previously made ourselves. This effect can already be seen in areas such as the clothing industry.

Software theft, commonly known as piracy, is the most prevalent form of copyright breach. Unlike the products of other creative processes, computer software is easily copied and distributed. Many people don't see software piracy as theft since software is not a visible article such as a book, a painting or sheet music. However, the penalties imposed on software pirates are very heavy, especially for the advertising and/or sale of infringing software. The main effect that piracy has is on the further development of software, since it deprives the legitimate copyright owners of money for further development. Piracy also causes an increase in the price of legitimate software, because of the need to spread the cost of development over a smaller number of purchasers.

Relationship between copyright laws and software licence agreements

Installation of a computer program will generally be accompanied by an agreement to abide by the terms of the software licence. The licence is a contract between the licence purchaser and the software developer to use the software in the manner specified in the licence. Since the licence is a legal document, a law (the Copyright Act) must back it up. The Business Software Association of Australia (BSAA) has been set up by software developers and distributors to help businesses comply with licensing and the copyright laws. The BSAA has a three-fold purpose: to educate management and employees, to assist with software management, and to help in the prosecution of those who breach their licence agreements. A more detailed description of the BSAA and its function can be found on the Internet.

Exercise 1.2

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

A software ________ describes the ________ placed on a user. When a licence is bought it allows the ________ to use the software, but the ________ remains the ________ of the developer. The licence will state the ________ of different ________ that can have the ________ installed on them as well as the rules for keeping a ________ copy.

2 Examine a number of software licences and list their common features. Using the features you have discovered, write a licence for a program you have written.

3 Explain the differences between reverse engineering and decompilation. Explain why most user licences forbid both decompilation and reverse engineering of the licensed software.

4 A small business buys a single-user copy of a word-processor program. A short time later, the business obtains a second computer. The owner decides to install the word processor on the second computer. Discuss the legal and ethical aspects of the owner's actions. What should the owner have done when the new computer arrived?

5 Explain why it is important for Australia to have a strong and well-enforced copyright law. Describe three possible effects of repealing (removing) the law of copyright.

The software market

Government, business and private individuals make up the software market. Some software types are common to all sectors of the market, for example operating systems, word processors and spreadsheets; others, such as games, are aimed more towards a particular sector. Government and business are more likely than individuals to require custom-written software, although many general-purpose applications can cater for the needs of businesses, especially smaller ones.

The software market is not driven just by the needs of the user. Many of today's common applications were the result of forward thinking by individual developers. We are fortunate that these people devised applications such as spreadsheets, desktop publishing software and integrated packages. The personal computer has become more widely used because of the development of these general-purpose software solutions.

The software market is also driven by the emergence of more sophisticated hardware and peripherals. As the capabilities of hardware increase, so does the sophistication of software applications. For example, early word processors had a limited number of formatting features available, because of the limited processing speed and capabilities of the CPU and relatively small primary storage. As the power of processors increased and more primary storage became available, the developers of word processors were able to incorporate more and more features into their products.

Maintaining market position

As with any business, the primary aim of a software development company is to make a profit. In order to remain competitive, a software manufacturer must maintain or improve its market position. In this respect, a software title is no different from a product such as a can of baked beans or a car.

There are two ways in which software developers can maintain their market position, first by improving the current applications and second by introducing new and innovative software applications.

Maintenance of and support for an existing product is very important for the survival of a software company. Some of the maintenance will involve the identification and correction of program faults. Some will involve improving the product by making the interface more intuitive or by adding new features.

Borland RAD Pack for Delphi
Rapid Application Development
Borland ReportSmith for Windows
The easiest way to spectacular reports!
Visual dBASE
Relational database for Windows
Borland 5.5
INTERBASE
The Scalable SQL Server
Workgroup Server
Borland 4.0

Figure 1.8 Software developers have to support existing software and develop new programs to keep their position in the market.

Most major software developers employ an active research and development program. The aim of research and development is to investigate new ways in which to use existing technology.

The effect on the marketplace

Software production is a multi-billion-dollar industry fuelled by consumer demands for the latest in technology. As developers bring out new applications or enhancements to existing software, consumers are enticed to update their applications. An upgrade to the new version is offered to existing users and often a crossgrade is offered to registered users of competing products. Software owners can often be persuaded to buy a licence for a new version of an application even though the new version offers no real benefits.

A side effect of the increasing complexity of software is the need for an increase in computer power to run the application, as well as a need for larger primary storage, larger secondary storage and more powerful output devices. Thus the sophistication of software can be thought of as driving the need for more powerful computers. This trend is probably most evident in game playing.

Exercise 1.3

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Both ________ purpose and ________ written software are available. Individual users are more likely to buy ________ software for their needs. Government and other ________ may need to commission ________ software for their specialised needs. Innovative software such as spreadsheets ________ and ________ were the result of ________ by individual developers. Other factors that drive the software market are ________ and ________ .

2 Explain, using an example, how a software developer can maintain market position over a long period of time.

3 Describe, in your own words, the factors that drive the software market. What is the role of the software developer in meeting these demands?

4 Explain how the emergence of new technology gives rise to new software and how new software demands may drive the need for new technology. Give examples to illustrate your answer.

5 Examine two different versions of the same software package. List the major features of each. What features of the newer package are not present in the older one? Which of these new features, if any, do you think are essential to the operation of the program? Which features of the newer program, if any, do you think were not really necessary for the normal user of the program?

Significant social and ethical issues

Software is used in a large number of differing situations, each of which impacts upon people. The software developer has a social and ethical responsibility to ensure that people benefit from a new system without forgoing any rights they have held up until its introduction. We can gain some insight into the issues faced by individuals, organisations or governments by looking at some scenarios.

National and international legal action resulting from software development

As with all other areas of human endeavour, software development may result in conflict. These conflicts are often resolved in the courts. As previously seen, copyright is one area in which conflict may occur, for example with two competing software producers creating applications with a similar look and feel. However, copyright is not the only area in which there is the potential for conflict. We will briefly look at two cases, one involving copyright and the other a perceived attempt at creating a monopoly.

The case of *Trumpet Software Pty Ltd & Anor v OzEmail Pty Ltd & Ors [1996] 560 FCA 1 (10 July 1996)* involved the distribution of a shareware program called *Trumpet Winsock*. *OzEmail* placed the installation program for *Trumpet Winsock* on an Internet setup disk that was distributed on the cover of a magazine. However, the need for users to register *Trumpet Winsock* separately from registering for the *OzEmail* service was not stated as part of the installation process. Some of the documentation that was supposed to be part of the shareware distribution was also omitted or changed, thus breaking the licence agreement. The judgment was in favour of *Trumpet Software Pty Ltd* with costs also awarded against *OzEmail*.

The United States government has spent a long time and a large amount of money in pursuing *Microsoft* in the courts over its attempts to monopolise the personal computer market with the *Windows* operating system and its *Internet Explorer* web browser. One of the main thrusts of the case is that, by including *Internet Explorer* with the *Windows* operating system, *Microsoft* is closing off the market to rival web browsers. The legal action has taken the form of an anti-trust case. At the time of writing, the judgment has gone against Microsoft, but the company is appealing the decision.

Current cases can be found by using one of the many legal search engines on the Internet.

Figure 1.9 Many software disputes are resolved in the courts.

Public issues

Decisions made by software developers can often have a widespread effect on individuals, either directly or indirectly. The following examples look at both indirect and direct impacts on the individual.

The year 2000 problem

Probably the most costly programming decision ever made was the one to represent the year by only two digits. The decision was made in the early years of commercial computer use when storage was expensive. The decision to reduce the representation of the year from four to two digits certainly saved a great deal of money when it was done. At the time it was thought that it would not matter very much as the software being developed was not anticipated to last until the year 2000. During the mid to late 1990s, the effects of this decision were realised, and money and effort were increasingly spent to overcome the problems that a misreading of 00 for 1900 instead of 2000 would cause. In Australia alone, it has been estimated that $12 000 000 000 was spent on overcoming the problem. Organisations felt the greatest effect from this problem, but individuals too had to face the possibility of disruption, which fortunately did not occur.

Figure 1.10 The beginning of the year 2000 was a worrying time for many people.

Computer viruses

Computer viruses are a malicious attempt to disrupt or destroy computer systems. Whatever the reasoning behind the writing of viruses, they cause a great deal of expense and effort either in prevention or in fixing the problems that they have caused. Preventative measures include the use of virus detection software, care in the transfer of files and the avoidance of the use of illegal software. Individuals and organisations need to ensure that they take the appropriate measures to avoid infection and the spread of viruses.

Figure 1.11 Common household appliances such as the microwave oven rely on software for their operation.

Reliance on software

Computer technology has invaded most aspects of life in the developed world. People are becoming more reliant on computer solutions to problems. Whether washing the dishes or enjoying a day out, we often use computer technology without knowing. The use of such technology brings with it a great responsibility for developers, as reliability may be critical. Software failure may cause only minor discomfort, as, for example, when an air-conditioning system fails, or it may be disastrous, such as in cases where human life is endangered.

Exercise 1.4

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

The year 2000 problem occurred because ________ chose to use ________ digits to represent each year instead of ________. This was all right until the year changed from ________ to 2000. If left untouched, many computer ________ would have read the year 2000 as 1900. Calculations which ________ the year would have resulted in ________. The problem was caused by ________ storage in the ________ days of computers, when the decision was made to represent the ________ by only ________ digits to reduce the amount of ________ needed. Programmers thought that the ________ they were writing would be ________ before the year 2000.

2 What is meant by the term computer virus? In what ways can a software developer help in avoiding the spread of viruses?

3 Explain the steps a software development company could take to avoid legal action if it wanted to incorporate a shareware application in one of its software releases.

4 Keep a scrapbook, or other record, of social and ethical issues that appear in the media or on the Internet. The newspaper websites can be used to help compile your record.

Team Activity

Develop an appropriate code of conduct that you can use. The code should address acknowledgment of authorship, ethical behaviour and an appropriate response to problems that may occur during the development cycle.

Review exercises

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

A virus is a small __________ that is spread by e-mail, __________ or __________ software. Software __________ have a responsibility to __________ the spread of viruses. Methods used include the __________ of antiviral __________ and not using the __________ computer for __________ use or __________ it to the Internet. Other sources of infection are __________ copies of programs and __________ storage media.

2 Choose the alternative, A, B, C or D, that best answers the question.

a Ownership of intellectual property in Australia is governed by
  A the laws of theft
  B copyright
  C the *Copyright Act 1968*
  D the International Copyright Law

b A software developer has a responsibility to
  A create reliable software
  B claim ownership of the software
  C quickly solve problems
  D follow the Software Development Code of Conduct

c A software developer took some work home to finish it on the children's computer. This should not have taken place because
  A the hard disk on the children's computer may fail, losing all the changes
  B there is a risk of picking up a virus from the home computer
  C the developer may spend all the evening playing games
  D the developer might not be able to use the computer, as the children may need it

d A software company will upgrade a software application when
  A the users want more functions
  B a rival company produces a similar program
  C all new computers use speech recognition for input
  D problems are found in the existing application

e A single-user software licence will allow the purchaser to
  A make copies of the program for friends
  B use the program at home and at work
  C make a backup copy of the program that is kept but not used
  D load the program on a small home network

3 Explain what is meant by the term 'decompilation'. Explain when decompilation of a program is legal, using an example.

4 Describe, in your own words, the rights and responsibilities of software developers.

5 'Decisions that are made by software developers can have a wide-ranging effect on the community.' Discuss this statement, using examples to help with your explanations.

Chapter summary

- Copyright allows the creator of an intellectual property to be acknowledged and compensated for it.
- A software licence is a legal contract that specifies how the software may be used.
- Software developers, as well as having the right of acknowledgment, have a responsibility to ensure that they produce quality software in an ethical manner that avoids the spread of viruses.
- Sources of previously written code need to be acknowledged and permission to use it must be given by its author.
- Software should be reliable, that is, perform without failure.
- A developer should ensure that the software produced, together with its documentation, is of the highest quality.
- Problems that occur in the development process should be overcome in an orderly and structured manner.
- Software developers should develop a code of ethical conduct and follow it.
- Developers must take all the precautions necessary to prevent the spread of viruses.
- Software theft, commonly known as piracy, deprives the copyright owner of income and violates the laws of copyright.
- Intellectual property is the result of a creative process.
- Plagiarism is the false claiming of ownership of an intellectual property.
- Software distributed free of charge is known as shareware if a payment is required after an initial trial period of use.
- Software that is distributed freely and which can be used without further payment is known as public domain software (also known as freeware).
- Copyright of public domain software still resides with the author.
- When software is purchased, it is a licence to use only the software that is being bought.
- The *Copyright Act 1968* governs copyright within Australia.
- The process of reverse engineering involves examining an existing system and using the understanding gained to create a similar system.
- Decompilation is the process of turning executable code into assembler language so that its structure can be more easily understood.
- The conditions in a software licence explain exactly how the software may be used.
- A licence will explain whether or not software can be used on a network.
- Enforcement of copyright helps to ensure the support and enhancement of existing software as well as the development of new software.
- A software licence is a legal contract between the purchaser and the software company.
- The software market consists of all those who want or need to use software.
- The developments in the software market are driven by improvements to existing software, the need for new software and the results of research and improvements in technology.
- From time to time, software issues can end up in a court of law. These issues may be as diverse as breaches of copyright through to anti-monopoly cases.
- Issues that result from software use affect individuals both directly and indirectly. Viruses and the Year 2000 problem have affected large numbers of people, and there is an increasing dependence on computers for everyday living.

chapter 2
Application of software development approaches

Outcomes

A student:

- differentiates between various methods used to construct software solutions (H 1.2)
- explains the relationship between emerging technologies and software development (H 2.2)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- applies appropriate development methods to solve software problems (H 4.2)
- applies project management techniques to maximise the productivity of the software development (H 5.1)
- creates and justifies the need for the various types of documentation required for a software solution (H 5.2)
- selects and applies appropriate software to facilitate the design and development of software solutions (H 5.3)
- assesses the relationship between the roles of people involved in the software development cycle (H 6.1)
- communicates the processes involved in a software solution to an inexperienced user (H 6.2)

Students learn about:

Software development approaches

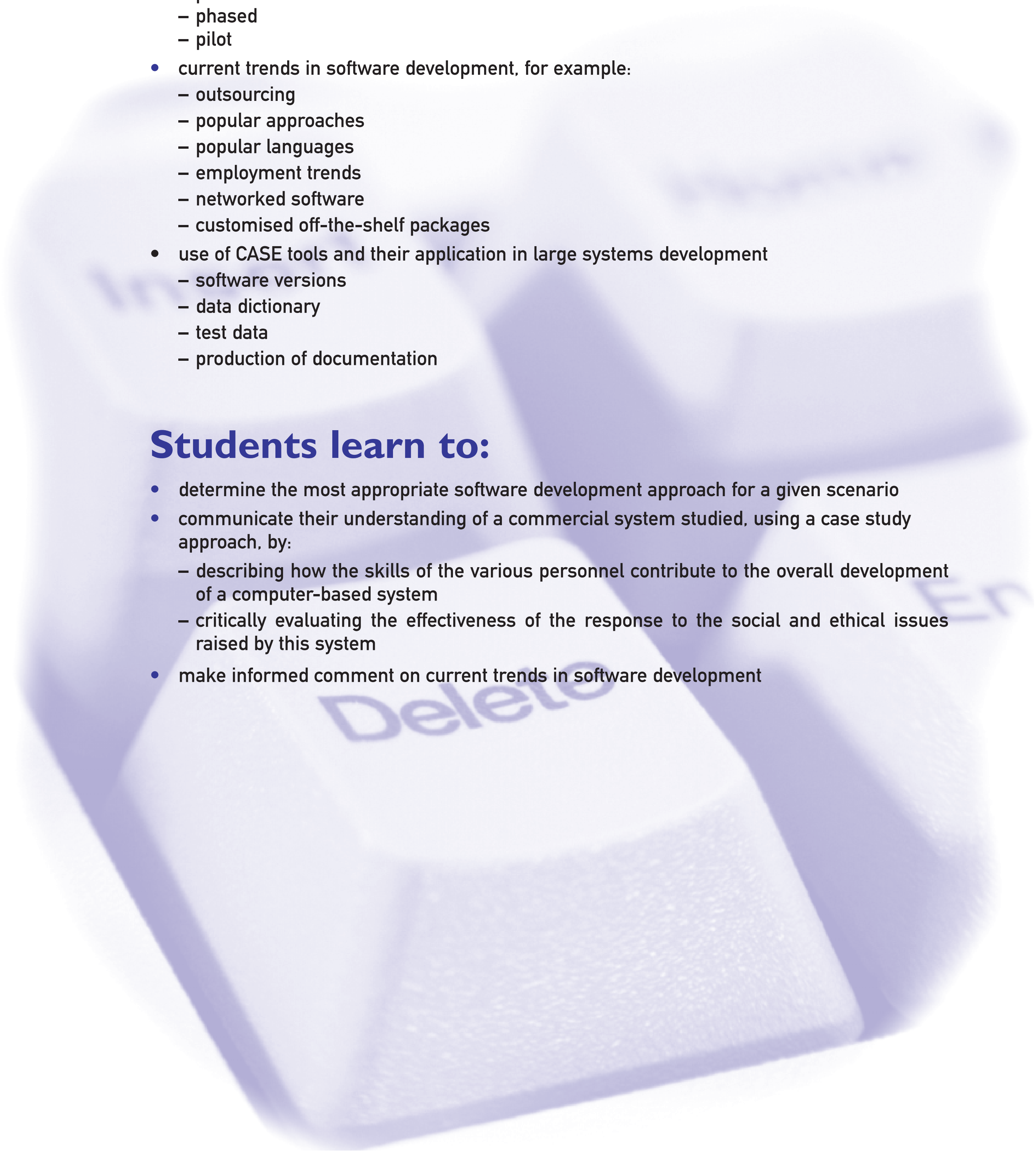
- approaches used in commercial systems, including:
  - the structured approach
  - prototyping
  - rapid applications development

– end-user development
– combinations of any of the above

- methods of implementation
  – direct cut over
  – parallel
  – phased
  – pilot
- current trends in software development, for example:
  – outsourcing
  – popular approaches
  – popular languages
  – employment trends
  – networked software
  – customised off-the-shelf packages
- use of CASE tools and their application in large systems development
  – software versions
  – data dictionary
  – test data
  – production of documentation

Students learn to:

- determine the most appropriate software development approach for a given scenario
- communicate their understanding of a commercial system studied, using a case study approach, by:
  – describing how the skills of the various personnel contribute to the overall development of a computer-based system
  – critically evaluating the effectiveness of the response to the social and ethical issues raised by this system
- make informed comment on current trends in software development



Preliminary review

No single correct software development approach is suitable for all problems. A number of different factors will influence the way in which a software package is developed. Time, budget and resources, the nature of the problem and the expertise of the developer will all influence the approach taken. The four main approaches are the structured approach, prototyping, rapid application development and end-user development. A combination of two or more of the approaches may also be employed, again dependent on the nature of the problem or problems to be solved.

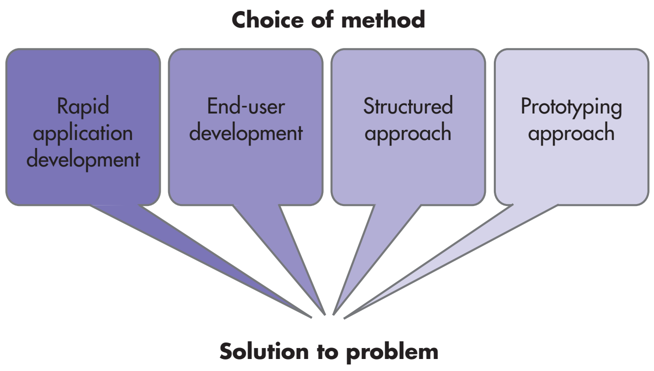


Figure 2.1 The four common approaches to software development may be used individually or in combinations.

The *structured development approach* is based on a tried and tested design method. It is divided into five stages: defining the problem, planning the solution, building the solution, checking the solution and modifying the solution. The structured approach is generally used for large projects that are being undertaken by a team of developers. The structured approach is not necessarily the best option, even for large projects, especially when time is critical, as each step must be completed before the next is undertaken.

Prototyping involves the creation of a working model of the system. The model may be used simply to gather information that can be used for another development approach, but it may also be developed into the final solution. Prototypes are created, then tested and evaluated, modified and tested again. This development approach is particularly appropriate for interactive software. It is not suitable for complex systems or those requiring complex mathematical manipulations. Although it is tempting to develop prototypes without appropriate documentation, this practice should be avoided, as maintenance of the solution will need the process documentation created.

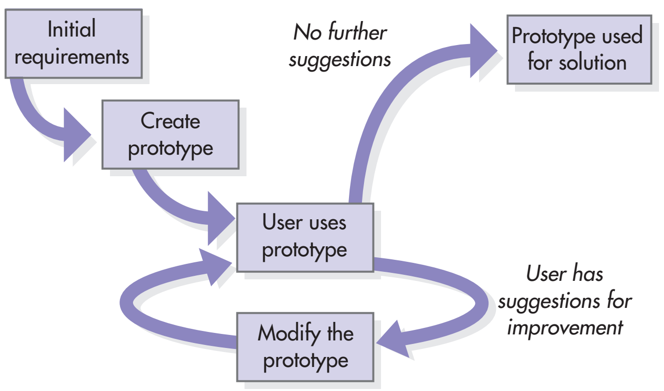
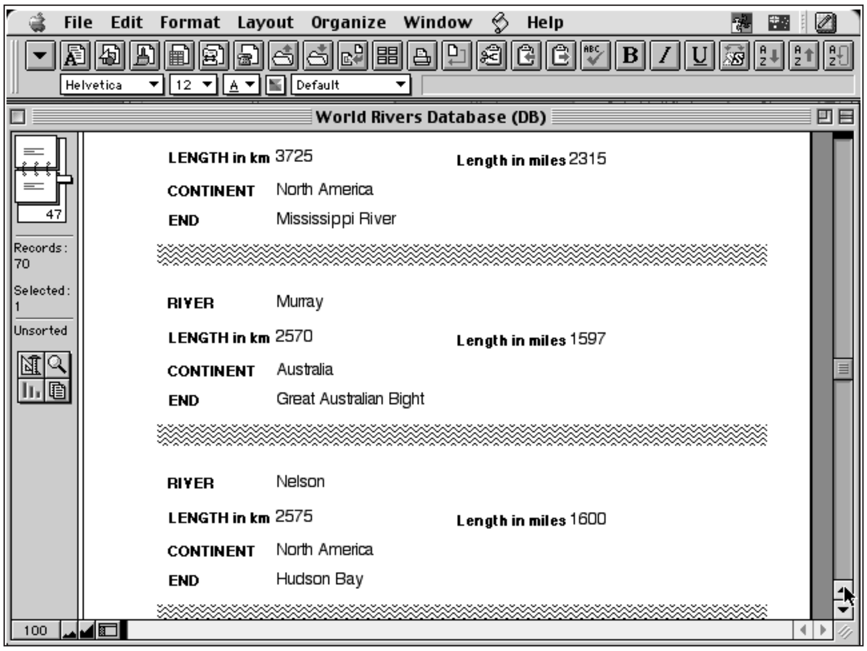


Figure 2.2 Prototyping involves a cycle of steps that are followed until the prototype performs all the required tasks in the appropriate manner.

Rapid application development (RAD) is a generic term used to describe any approach that leads to faster application development. Some of the RAD approaches available are the re-use of code, the use of computer-aided software engineering (CASE) tools and the use of templates. Development environments such as Visual Basic, Hypercard, Hyperstudio, Microsoft Access, Filemaker Pro and REAL Basic are examples of CASE software tools. The RAD approach usually lacks formal stages and is more suited to small, low-cost projects with small development teams. The user is often directly involved with the programmer when an RAD approach is taken.

End-user development generally involves the adaptation of software tools by the user. End-user development takes a very informal approach to the development process. It is used for very small solutions that can be created at a fraction of the cost of a custom-built software package. Most end-user development occurs in a fourth-generation programming environment such as a spreadsheet or a database management system.


File Edit Format Layout Organize Window Help
Helvetica
12
Default
World Rivers Database (DB)
47
Records: 70
Selected: 1
Unsorted
100

LENGTH in km 3725 Length in miles 2315
CONTINENT North America
END Mississippi River

RIVER Murray
LENGTH in km 2570 Length in miles 1597
CONTINENT Australia
END Great Australian Bight

RIVER Nelson
LENGTH in km 2575 Length in miles 1600
CONTINENT North America
END Hudson Bay

Figure 2.3 A database management system will often be used to create an end-user software solution.

Exercise 2.1

1. Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

   The four main approaches to ________ development are ________ , ________ , ________ and ________. The choice of a development method will depend on a number of ________ including the ________ of the problem, the ________ of the ________ , time, ________ and resources available.

2. Name and briefly describe the five stages in the structured approach to software development.
3. When is the structured approach to software development most suited to a project? Give reasons for your answer.
4. What is a prototype? Briefly describe the process of prototyping.

5 Explain why documentation is important in the prototyping process.

6 Explain the meaning of the term 'rapid application development'. Describe the types of application development that are suited to this approach.

7 What types of project are suited to end-user development? Explain why this approach would be used in these projects.

8 An airline is installing touch screens for passenger entertainment in its aircraft. Which of the software development approaches do you think is most suited for this project? Justify your answer.

9 Use an appropriate computer application to create a table that lists the features of each of the software development approaches. Save your file for future reference.

10 Investigate the features of one application that can be used by an end-user developer. Describe, in general terms, the types of project that this application is suited to.

Software development approaches

As discussed, there is no one approach that can be taken for all software development projects. Each project has its own needs; some require a large team of developers; others can be undertaken by an individual. In all projects, however, it is important to carefully document the approach taken, as this will help with maintenance of the project and with future development of similar projects.

The software approach that is taken to complete a project will depend on a number of criteria. The most obvious criterion is the *size* of the project. A big project taking a large team a long time to develop will need a far more structured approach than a small project being undertaken by the end user. However, modules of a large project may be created using one or more of the alternative approaches. A small end-user project may, rightly or not, be an ad-hoc affair with very little documentation. Projects that involve a great deal of user interaction, with very little complex processing, are likely to be suited to a prototyping approach, using feedback from users to refine the final model. Common small, but specialised business tasks can often be completed using an RAD approach.

Other factors that will affect the development approach taken are the *time* available for implementation and the *skills* of the development team. Development projects that have to be completed within a particular time frame can force the development team into "shortcuts", such as the use of CASE tools to assist the process. The skills of the development team may prompt them to use a particular development approach; this is especially true for small teams of developers whose total experience will probably not cover all development approaches.

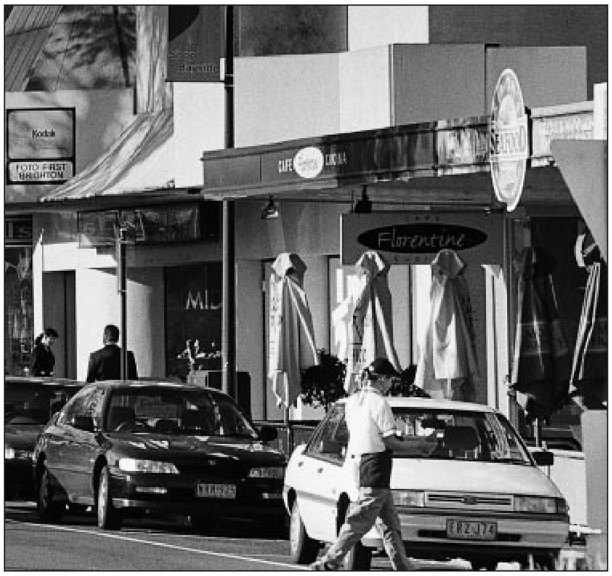


Figure 2.4 Small businesses can often customise application software to perform the required tasks.

Approaches used in commercial systems

The structured approach

The structured approach to software development, as its name suggests, is the most formal and structured of the four approaches. Each of the phases of the development process is thoroughly documented so that a complete record can serve as a reference for the subsequent stages in the development process. The structured approach can involve a number of specialists. These specialists are usually categorised as systems analysts and programmers.

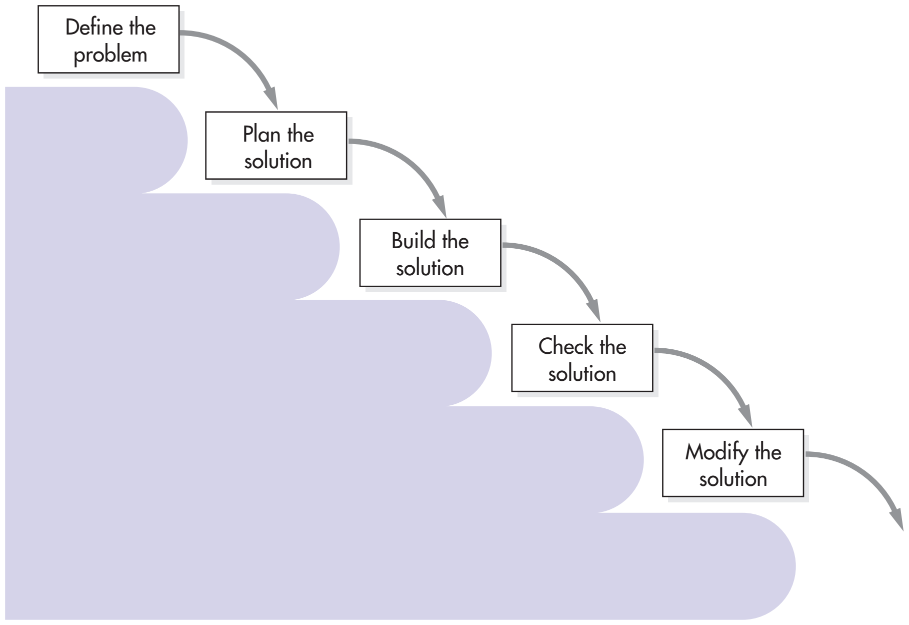


Figure 2.5 The structured approach to software development is often represented as a waterfall.

The first two phases are generally the responsibility of a *systems analyst*. A systems analyst is a person who develops a system to meet a new need or to solve an existing problem. This may involve investigating the current system and modifying it, or developing a new system. A systems analyst could be involved in the planning, implementation and maintenance of the computer-based system. As well as technical computing skills, the systems analyst must have good management and personal skills, in order to be able to interview users of the system, determine their hardware and software requirements and develop plans and schedules. Most systems analysts have university degrees in either computer science or business. They must have a broad knowledge of computers and keep up to date with recent developments in hardware and software. They are often required to test new products. Systems analysts are given a variety of titles, such as computer analyst, systems consultant or systems officer.

Programmers perform a large amount of the work carried out in the other stages of the structured approach. The programmer depends on the systems analyst to provide the detailed specifications of the required program. There are two types of programmer, the system programmer and the application programmer. A system programmer writes the instructions for the computer's operating system, whereas an application programmer writes the instructions for a computer to perform a particular task. Programmers usually need to complete a computer course at a tertiary institution. They are required to know several different programming languages and types of hardware.

Management and users are also involved in the development process, as they have the detailed knowledge about the existing system. It would be a very hard task indeed to create an effective program if management and the users did not have input into the development process.

The first step taken in the structured approach is to define the problem that is to be solved by the software. This may seem to be a fairly trivial stage, since if a software solution is needed then a problem has been identified. However, problem definition looks not just at the problems that are being experienced by users of the current system but at all aspects of the system. During the problem-definition stage of development, the team of systems analysts interviews management and the users, and collects data about the existing system by observation, collection of documents, questionnaires and surveys. Once the data has been collected and analysed, a set of specific requirements is drawn up. The requirements are used to design the software and as a measure of the performance of the final product. The problem definition will also state any restrictions that may be placed on the software solution.

A further important aspect of the problem definition is to determine the feasibility of a software solution. There are many factors that need to be taken into consideration when determining if it is feasible to proceed with the development of a new software solution. Factors that affect the feasibility of a software development include: whether the problem is worth solving, constraints on the development, financial feasibility, operational feasibility, technical feasibility, scheduling, possible alternatives, and the social and ethical considerations.

Constraints on the development are factors that limit the development in any way, such as the amount of change needed to implement the new system.

Financial feasibility is concerned with the ability of the organisation to profit from the implementation of the proposed system. The financial feasibility of a new system is often presented as a *cost–benefit analysis* in which the cost of the new system is compared with the benefits.

Operational feasibility is centred on the capability of the organisation to handle the manual processing involved with the system.

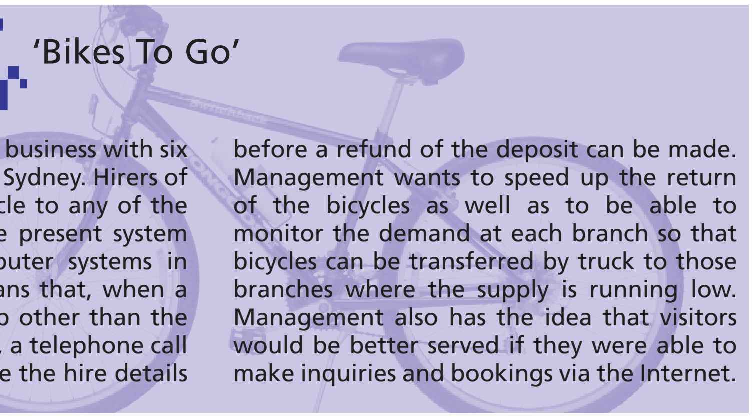
Technical feasibility is concerned with matching the organisation's technical resources with those needed for the new system. It also involves an assessment of the required hardware and software to ensure that the tasks can be carried out by the proposed system.

Scheduling feasibility examines whether the proposed system can be implemented within a deadline. The deadline is often inflexible because of some external influence; for example, the Year 2000 problem had a deadline of 1 January 2000 which could not be postponed to allow for a longer development time.

Any new system needs to be developed with social and ethical considerations. For example, if the new system involves amassing a large database of personal information, the safeguards for the individual need to be fully investigated.

Case study

'Bikes To Go'



'Bikes To Go' is a bicycle hire business with six outlets in various suburbs of Sydney. Hirers of bicycles can return the bicycle to any of the outlets around the city. The present system consists of individual computer systems in each of the shops. This means that, when a bicycle is returned to a shop other than the one from which it was hired, a telephone call has to be made to determine the hire details before a refund of the deposit can be made. Management wants to speed up the return of the bicycles as well as to be able to monitor the demand at each branch so that bicycles can be transferred by truck to those branches where the supply is running low. Management also has the idea that visitors would be better served if they were able to make inquiries and bookings via the Internet.

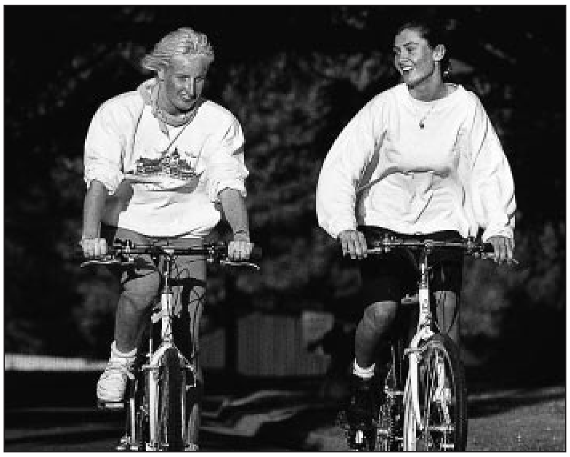


Figure 2.6 Cyclists can hire a bike at one store and return it at another.

With these requirements in mind, management contracted a development team to update the computer software system. After a preliminary meeting the team decided to employ a structured approach for the development process, as the required system is quite complex. However, they have left open the possibility of using different development approaches for each of the major software components.

The first phase of the development process, defining the problem, involves the systems analysts, management and the users of the system. An overall set of requirements is gathered in initial meetings with management and the users in order to gather information about exactly what the system is going to do. The result of this activity is a report detailing the requirements, together with a feasibility study.

In order for the system to work effectively, its operation needs to be fully understood. The next phase in the process is for the analysts to gather information about the present workings of the system. The information is gathered in a number of different ways, including interviews with staff and customers, observation of the processes that are carried out and an examination of the documents that form part of the system. The results of this phase are presented as a report. Aspects of the system are often represented diagrammatically in the form of a dataflow diagram, or in a summary form such as an IPO chart.

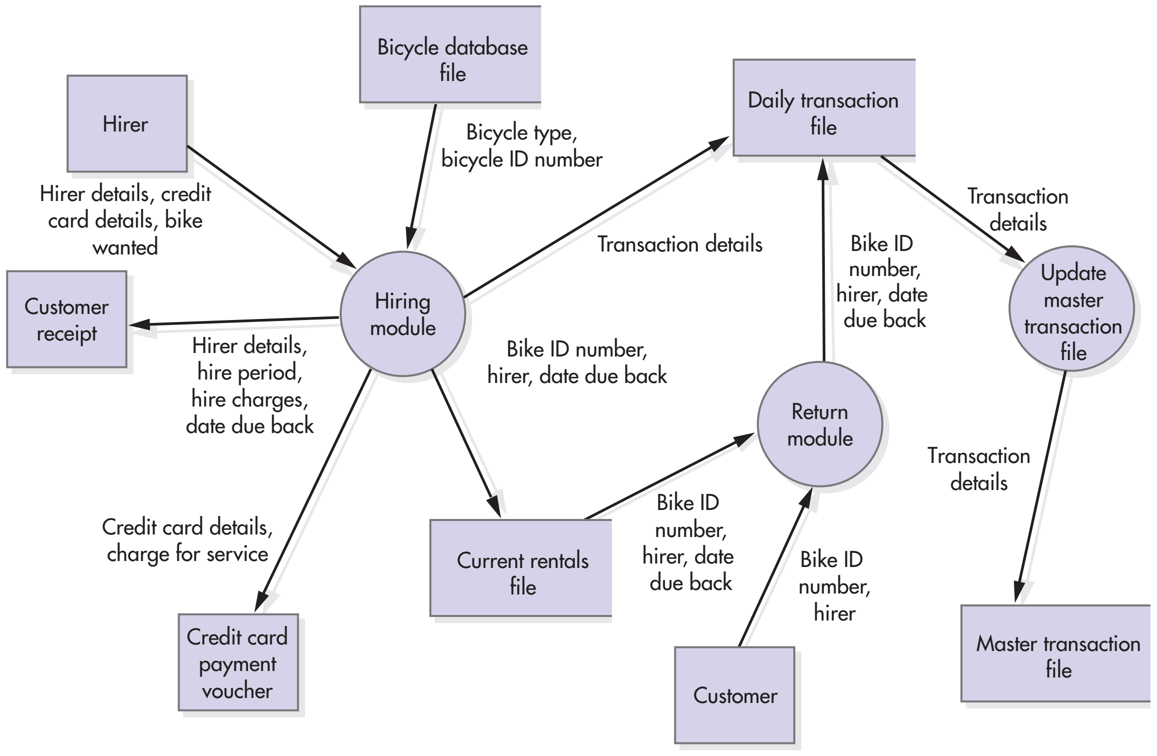

Bicycle database file
Hirer
Daily transaction file
Bicycle type, bicycle ID number
Hirer details, credit card details, bike wanted
Transaction details
Bike ID number, hirer, date due back
Transaction details
Customer receipt
Hiring module
Update master transaction file
Hirer details, hire period, hire charges, date due back
Bike ID number, hirer, date due back
Return module
Transaction details
Credit card details, charge for service
Current rentals file
Bike ID number, hirer, date due back
Bike ID number, hirer
Credit card payment voucher
Customer
Master transaction file

Figure 2.7 A dataflow diagram representing the 'Bikes To Go' system.

Input	Process	Output
Hirer name Hirer address Bicycle type Bicycle identification number (BIN) Hire period Hire rate Deposit taken Credit card type Credit card number	Input customer details, hire period required and credit card details (if applicable). Input bicycle type. Allocate available BIN from *bicycle* file. Calculate the date and/or time the bicycle is due back. Look up the hire rate in the *rate* file. Calculate the total charge for bicycle hire. Generate customer receipt. Generate a credit card debit form if required. Write the transaction details to the daily transaction file.	Customer receipt with hirer details, bicycle type and BIN, the hire period, date or time the bicycle is due back, hire rate, the total charge and the amount of deposit. Credit card debit form with the card details and amount. Transaction details to a daily transaction file.

Figure 2.8 An IPO chart representing the hiring system.

The second stage of this development approach involves planning the solution. Planning involves choosing the appropriate data structures and program structure, planning the characteristics of the user interface and designing the algorithms. During the planning stage, decisions will be made as to the most appropriate language or languages for the construction of the solutions. The planning stage may also involve scheduling the project so that it can be completed within a reasonable time-span.

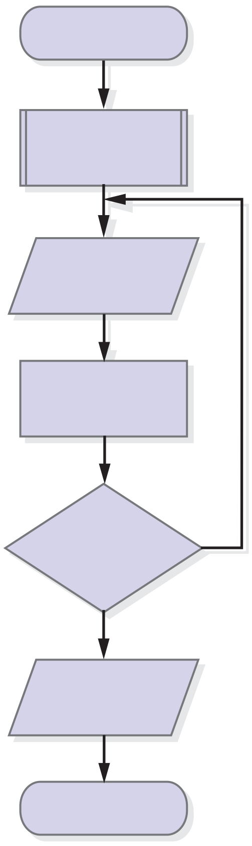
In the case of 'Bikes To Go' it was decided to break the main program down into three modules: one to process the hiring, one to process the returns and the third to look after the administration of the shops. The third module needed to be able to maintain the master transaction file, maintain the bicycle database and perform the necessary accounting duties. Two other modules were also deemed necessary for the solution of the problem, an initialisation module and a closing module.

The systems analysts, together with the programming team, met to plan the overall design of the program. At this stage, it was decided to concentrate on a solution to the problems, leaving the Internet facility for a later date. The development team was divided into five teams, each with responsibility for one of the modules. A set of test data items was created for use in the later stages. Before the teams began work on their individual projects, the form of common data items was determined, as were the standards that were going to be used in the interface design. Each then concentrated on the design of the assigned module. At this stage, this mostly involved the design and checking of the algorithms and interface.

The building stage of the development process is the one most often thought of as 'programming', the coding and testing phase. In this process, the algorithms are converted into machine executable form and tested. Depending on the development environment that has been chosen, the machine executable code may be an object file or it may be a file such as a spreadsheet. The program may be built in a 'top down' fashion where the driver module is first constructed and the sub-modules written first as stubs, or in a 'bottom up' fashion, where the sub-modules are written first and then gradually combined to form the whole application.

Figure 2.9 One of the major parts of the planning stage in the structured approach is the design of the algorithms.

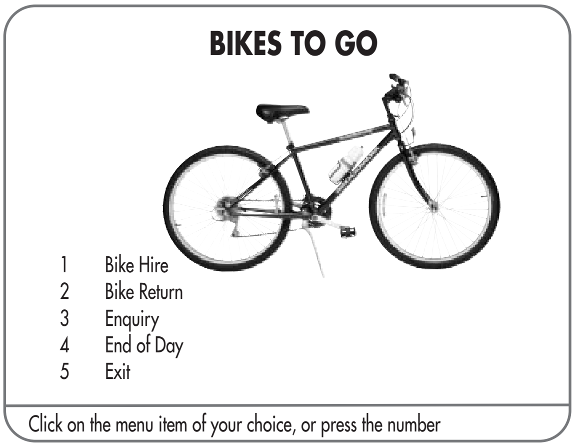
```
BEGIN
        set counter to one
        set sum to one
        set number to user input
        REPEAT
            add counter to sum
            increase counter by one
        UNTIL counter equals number
        print sum
END
```



In the case of 'Bikes To Go' the programming team decided to use a top-down approach to the development of the program. The driver module first constructed for the system was basically a sequence that called upon three sub-modules: an initialisation module, a menu module and a closing module. Each of the five teams was assigned responsibility for one of the modules. The senior programmer took the overall responsibility for co-ordination of the project. During this stage of development, regular meetings were called to gauge the progress of development and to discuss any problems that occurred. Towards the end of this phase, a number of peer checking and structured walk-through activities were also undertaken. Once the full program had been assembled, it was ready for the next stage of development, the checking stage.

Once the full program has been assembled and tested, it is time for it to be checked. In this stage, performance is measured against the original specifications. This process is known as **acceptance testing** if employed on custom software or **beta testing** when used during application software development. At this stage, real data is used, with the outputs being compared to the real outputs from the previous system. Users are also involved in this process to ensure that their expectations of the system are fulfilled. In addition to the feedback that the development team receives, users also benefit by being able to use the system. In addition to checking that the program can process the data correctly, the user interface is evaluated for ease of use and intuitive design. The manner in which the program responds in various situations, for example the manner in which errors

Figure 2.10 The programmer's task in the building stage is to code the algorithms.



BIKES TO GO

1 Bike Hire
2 Bike Return
3 Enquiry
4 End of Day
5 Exit

Click on the menu item of your choice, or press the number

Figure 2.11 'Bikes To Go' now has a fully operational program.

are handled, is also evaluated. Specifications for any necessary modifications are created during this phase of development. Once modifications have been made and tested, the program is either accepted or modified further. This process of testing and modification continues until the program is accepted for use.

'Bikes To Go' involved one employee from each of the stores in the checking process. In this way, not only does testing take place but also staff training. The data used by 'Bikes To Go' and the development team consisted of one month's actual rental figures. Apart from a few suggestions in regard to the user interface, the testing process found no problems with the processing of the real data. Once the modifications to the user interface were made, the program was accepted for use by management.

Prototyping

Prototypes can be used in two different ways during the development process. The first is to use a prototype as an information-gathering tool only; the second is to develop the prototype into a fully working program. Prototyping as a development approach is especially useful for applications involving a lot of human–computer interaction, such as multimedia and Internet applications. The process involves the construction of an initial model of the program based on the specifications furnished by the client. Once the initial prototype is constructed, it is passed on to the user for evaluation. Based on the results of the evaluation, the prototype is refined. This process of evaluation and refinement continues until the prototype performs the required tasks within the bounds of the specifications.

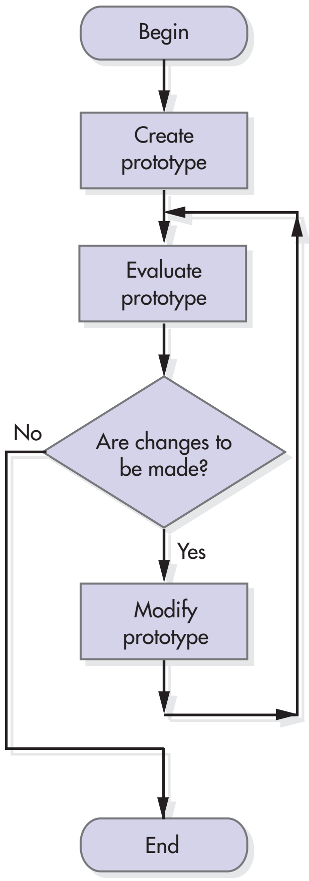


Figure 2.12 Evolutionary prototyping involves a cycle of constructions and evaluation until all the specifications and user requirements are met.

Case study 'Bikes To Go' website development

In the previous section we saw how 'Bikes To Go' developed its management program. After a few months of operation, management of 'Bikes To Go' decided to implement the second phase of their program update by commissioning the design of the Internet site to link in with their main program. The site is to be integrated with their hire database to allow potential customers to look at bicycle availability and charges. Regular customers are to be given password-protected access to pre-booking and pre-payment for hire. The web page is also to be provided with a number of suggested tours, each of them accompanied by a map and points of interest.

The development team that originally designed and implemented the hire program was engaged to develop the website. They decided to employ an evolutionary prototyping approach to development. After consultation with management about the requirements of the webpage, they set about designing a prototype. The prototype was set up on a single personal computer and then used in simulated situations, first with management and the development staff acting as users, then with a number of regular customers of 'Bikes To Go'. At each stage of the prototyping process, documentation was kept that showed the problems encountered and the solutions found.

Within a relatively short period of time, 'Bikes To Go' was able to make the site available for use.

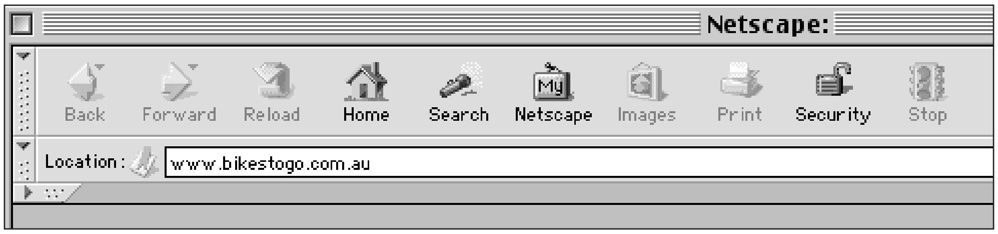


Figure 2.13 'Bikes to Go' was able to greatly improve its exposure by the use of an Internet site.

Rapid application development

Rapid application development (RAD) is used to speed up the development process. It is especially useful in cases where the requirements of the system are well understood and the project scope is well constrained. The RAD approach relies on the reuse of existing modules. Often these modules are supplied as part of a fourth-generation language. By using pre-existing modules, the need to test their working is avoided, thus reducing development time. If modules have to be written, they are constructed as reusable components. RAD relies on the use of automated tools to help in this construction.

Case study — Neumann's Nurseries

Neumann's Nurseries is a commercial nursery that grows plants for the wholesale market. The company has a number of greenhouses that are used to raise seedlings. At present the environment in these greenhouses is being controlled manually. Because of the difficulty of keeping the temperature, humidity and soil water content within acceptable bounds, management has decided to install a computer system to monitor and control the environment. A small software development company has been contracted to write the computer program that will oversee the system.

Since the program is to interact with sensors to control the environments, it was decided that an object-oriented programming environment was the most suitable to develop this software. The choice of object-oriented design aided by an RAD approach meant that the system could be introduced during a period of low demand. A further advantage of the object-oriented approach was that it allowed the nursery to change either the sensors or the actuators without needing to reprogram for the new hardware.

During development of the software, a number of pre-existing modules were used as a basis for the program. Modules already exist that can be used to gather data from temperature and moisture sensors and to send messages to the actuators that would turn on the heating, water, cooling and misting hardware when necessary. The developers were able to use this basis to quickly create a working program that fully met the requirements of the nursery.

Figure 2.14 The use of computer technology in commercial greenhouses can improve the quality of plants produced.

End-user development

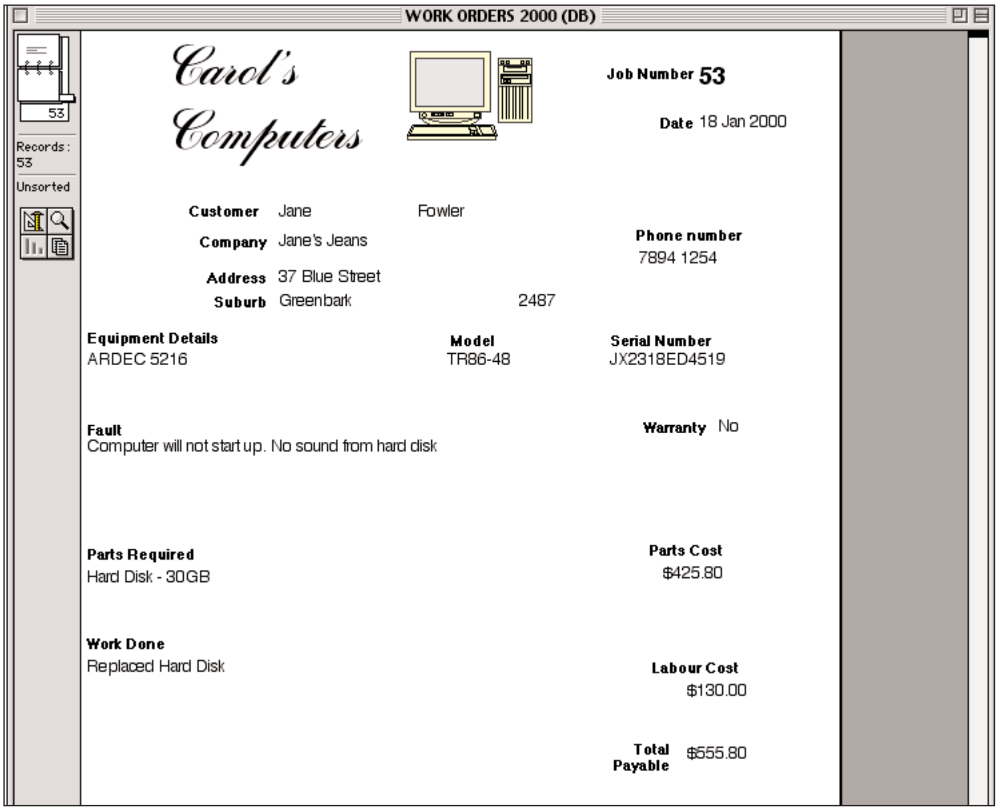
End-user development is probably the most popular of all development methods since most computer users have, at one time or another, created a solution for a problem. The problems solved are usually small and personal ones. Small business operators will also often use an end-user approach. Most of these solutions have been created using an integrated package, a spreadsheet application or a database management system. Development usually follows a fairly informal approach with little, if any, documentation.

Case study

Carol's Computer Repairs

Carol is a computer technician. After a number of years working for someone else, she decided to start a computer repair business herself. While working through her business plan, she realised that she would need a reliable system to manage the repair documentation.

All she needed was a system that could record the details of items brought into the shop for repair, then produce an invoice for payment. As she already had a computer that was loaded with an integrated package, she decided to create a system that she could use for the business. She used the database management system of the package to create a book-in sheet that could also be used as an invoice. Within a very short time she had a workable system. As her first few weeks in business progressed, she gradually refined the system to overcome some shortcomings.



Job Number 53

Date 18 Jan 2000

Customer Jane Fowler

Company Jane's Jeans

Phone number 7894 1254

Address 37 Blue Street

Suburb Greenbark 2487

Equipment Details	Model	Serial Number
ARDEC 5216	TR86-48	JX2318ED4519

Fault
Computer will not start up. No sound from hard disk

Warranty No

Parts Required
Hard Disk - 30GB

Parts Cost
$425.80

Work Done
Replaced Hard Disk

Labour Cost
$130.00

Total Payable $555.80

Figure 2.15 A database management system is able to help end-users find a solution to a problem.

Combinations of approaches

As seen in the case of 'Bikes To Go', it may be necessary to use a combination of development approaches to solve a problem. Certain aspects of a solution may lend themselves more to one development approach than to another. The ultimate goal of the development is to produce a properly documented solution to a problem.

Exercise 2.2

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

A ________ is a person who develops a system to meet a ________ or solve an existing ________ . The ________ is generally responsible for the first ________ phases of the ________ approach. A large amount of the work carried out in the other stages is performed by ________ . There are two types of ________ the system ________ and the ________ programmer. Also involved in the structured approach are ________ and ________ who provide information about the ________ system.

2 Describe the role of the user in the first stage of the structured approach to development. Name the other stages of the development cycle that involve the user. What role does the user take in these stages?

3 A programmer needs a number of different skills. Describe these skills and state, for each skill, why you think that a programmer requires that skill.

4 Describe the benefits that the new Bikes To Go system will have for the customer. What benefits will the business have when the new system has been implemented?

5 Describe the differences between an information-gathering prototype and an evolutionary prototype. Explain why an evolutionary prototype was used for the 'Bikes To Go' website construction rather than an information-gathering prototype.

6 Explain the advantages that the RAD approach to software development provided for Neumann's Nurseries. Give reasons why a structured approach to software development was not taken in this case.

7 What are Carol's reasons for choosing an end-user approach when designing her software solution.

8 Explain, using an example, why it may be necessary to use a combination of approaches to develop a software solution. Your example should state the problem being solved and the approaches you would choose for the various modules.

9 Employ an end-user development approach for the following problem: You want to keep a record of your progress in each of your school subjects. For each subject you want to track your results in each of the assessment tasks. Your solution should be saved as a file with the name TASKTRAK. Explain why an end-user approach is suitable for this project.

Methods of implementation

Once all the components of the system have been properly installed, the implementation of the new system can proceed. This involves the conversion from any existing system to the new system. This conversion needs to be carried out in such a way as to allow a smooth change, without any loss of data, time or money, but also allowing for possible errors or problems that have not been identified and corrected to this stage. There are four methods generally used when converting to new systems. These are direct conversion, pilot conversion, parallel conversion and phased conversion.

Direct cut-over

A direct cut-over (or direct conversion) involves the complete and immediate conversion to the new system. The old system is not used from the time the new system is first used. All data stored in the old system would need to be converted for use with the new system. If the new system involves compatible hardware and new software, this conversion may be a minor job. But if computers are being installed for the first time, this becomes a major problem. In this case, old data may be entered with the help of extra staff, through the use of scanners and OCR software, or not entered at all. If the new system fails or problems become evident, under this type of conversion the old system is not available as a backup and so this method is not often used.

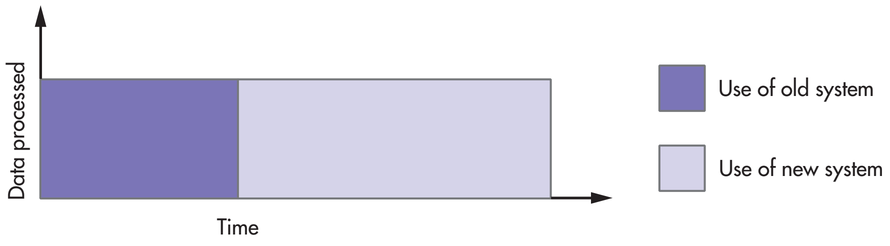


Figure 2.16 A direct cut-over is an immediate and complete conversion to the new system.

Parallel conversion

Parallel conversion involves the old and new systems both being used fully for a period of time. In this way, all operations of the new system duplicate similar operations in the old system, which allows the new system to be tested with a full set of data under realistic conditions, and also allows the two systems to be compared. If the new system fails, the old system can be used with minimum loss of data as it is kept up-to-date. A drawback of this method is that it creates additional workloads for those using both the new and the old systems. It also requires all staff to be trained before the implementation of the new system.

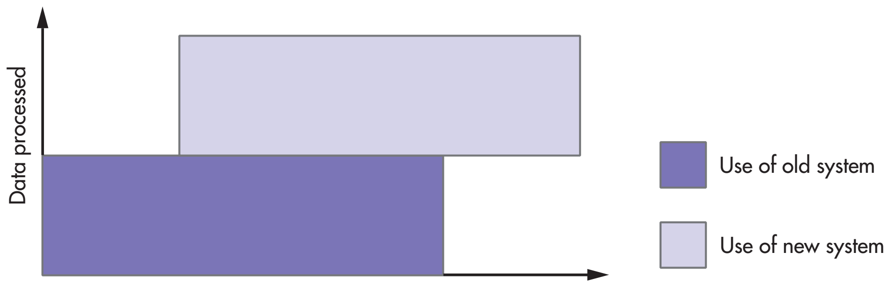


Figure 2.17 Parallel conversion involves the complete use of both systems together for a period of time.

Phased conversion

Phased conversion involves the gradual implementation of the new system to carry out certain operations while the old system is used for other operations. The operating modules of the new system can be used to test the system and to train staff. As modules are tested and operating and the staff become more confident, the new modules can be implemented until the whole system is operating. This allows each new module to be tested individually, without the threat to data stored in other modules or in the old system. If the new system experiences problems or fails, only that module is affected.

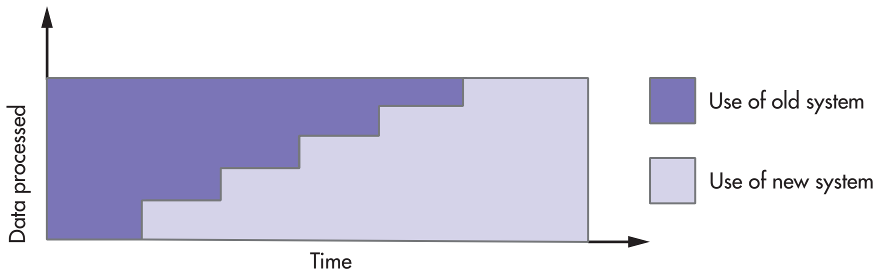


Figure 2.18 Phased conversion involves the gradual introduction of parts of the new system over a period of time.

Pilot conversion

In a pilot conversion the new system is fully installed but is used for only some operations as the old system is still being used. For example, in an office some of the staff may use a new word-processing system, while the rest of the staff use the old system. In this way the old system is still available if the new system fails or experiences problems. If the new system fails, only a small amount of data is lost as the old system is still maintained. It also allows the users to evaluate the new system as a complete system and to train others in its use, without the problems associated with direct conversion.

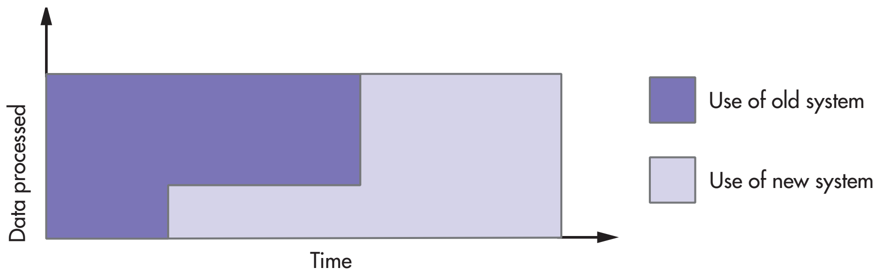


Figure 2.19 Pilot conversion involves the use of parts of the fully installed new system while the old one is still in use.

Exercise 2.3

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

There are four different methods of conversion: ________, ________ conversion, ________ conversion and ________ conversion. A ________ involves the complete and immediate conversion to the ________. ________ conversion involves the gradual ________ of the new system to carry out certain ________ while the ________ system is used for others. Full installation of the ________ system which is used for some of the operations, while the rest are performed by the old one, is known as ________ conversion. ________ conversion involves the full use of both the ________ system and the ________ system for all operations.

2 What are the disadvantages of direct conversion?

3 What would happen if the new computer system broke down under direct conversion? Is there any backup method?

4 What are the advantages of parallel conversion to the new system?

5 What are the disadvantages of parallel conversion?

6 What would happen if the new computer system broke down under parallel conversion? Is there any backup method?

7 What are the advantages of phased conversion to the new system?

8 What are the disadvantages of phased conversion?

9 What would happen if the new computer system broke down under phased conversion? Is there any backup method?

10 What are the advantages of pilot conversion to the new computer system?

11 What are the disadvantages of pilot conversion?

12 What would happen if the new computer system broke down under pilot conversion? Is there any backup method?

13 In the previous section you examined the 'Bikes To Go' program development. Which method of implementation would you recommend for the solution? Give reasons for your choice.

Current trends in software development

Software development itself is continually going through a process of evolution. The first development approaches devised were very formal, being based on the need to program computers in a way that was easy to convert into machine code. As the cost of computing power decreased, programming environments became more sophisticated, allowing programmers to employ different ways of programming. The decrease in the cost of computer power also made it possible for the general population to have access to the technology. Greater access also brought with it an increase in the number of non-computer professionals who were able to create programs for their own use.

The emphasis in industry and commerce has changed from a use of mainframe computers and terminals to networked personal computers. This change in direction has brought about a change in the way that software is obtained. In the early days of commercial computing, in-house specialists who formed a 'Data Processing Department' generally wrote software for a specific application. Two businesses with exactly the same needs and computer system would each write their own application. With the increase in the number of business computers, a greater standardisation of operating systems and the identification of common tasks, off-the-shelf applications have become a popular option to custom-designed software. For those applications that are more specific to a particular user, software is often obtained by contracting outside software development specialists or by end-user development.

Outsourcing

Many large organisations have decided to disband, or at least reduce the size of, permanent information technology staff. Where an information technology department does exist, it will often consist of those whose task it is to maintain the system. If new software is required, its development will often be contracted to outside developers. This is known as **outsourcing**. Outsourcing allows a business to have quality software developed by experts without the expense of maintaining a full Information Technology department.

Popular approaches

As discussed, there are a number of approaches to software development. These range from the very formal structured approach to the almost ad-hoc approach taken by many end-user developers. Software tools such as webpage editors encourage an evolutionary prototyping approach, especially from end-users. Commerce and industry often need to develop software that is accessible to a large number of employees. Employing a **client/server** approach to software engineering caters for this need. Small end-users often approach their software problems by customising off-the-shelf applications to provide the solution.

Popular languages

Languages used range from the formally structured software development environments of C++, Visual Basic and Java through to informal languages such as spreadsheets, webpage editors and macro recording tools. People with little or no formal training in computer programming can create quite sophisticated software solutions using the latter tools. As computer technology becomes more and more powerful and accessible, variations to these languages will become increasingly popular.

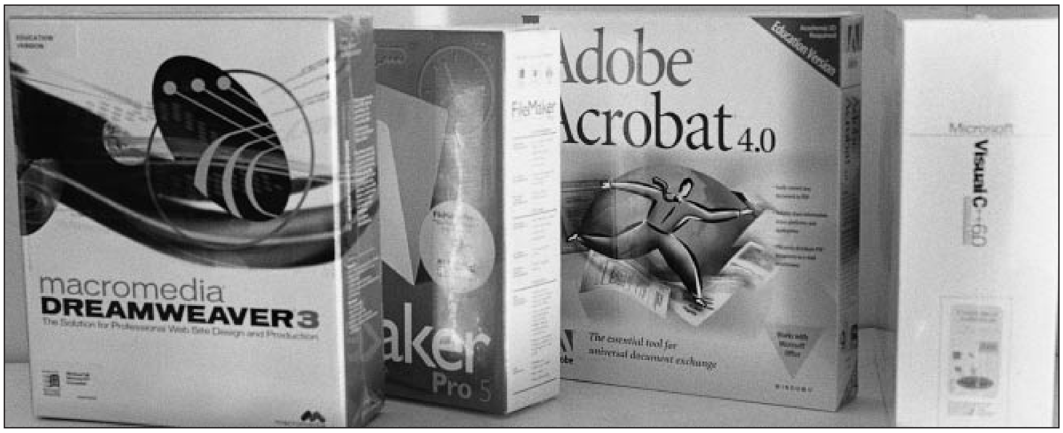

macromedia
DREAMWEAVER 3
The Solution for Professional Web Site Design and Production
FileMaker
Pro 5
Adobe
Acrobat 4.0
The essential tool for universal document exchange
Education Version
Microsoft
Visual C++ 6.0

Figure 2.20 Software development tools such as these are popular for end-user developers.

Employment trends

The move by business to outsource software development means that the employment of development staff is less likely to be permanent. The trend is for software developers to be contracted to either the business requiring the software or a software development company. Thus those in employment in the area of information technology are most likely to be working either under a contract that lasts until the end of development or for a fixed-term contract with a development team.

Networked software

As noted above, the trend for business is to move away from time-sharing a mainframe computer to a network of personal computers linked to a server. This trend in computer management also impacts on the development of software. The development of client/server software involves four parts: the interface, the application, database management and the network software. The interface governs the interaction between the user and the system; it must communicate not only the state of the application but also network messages. The application is responsible for meeting the software requirements of the end-user. The database management section is responsible for the manipulation and management of data. For example, the database management part will be responsible for processing inquiries. The network software provides the means by which communication takes place on the network. The developer not only has to create the software to meet the requirements of the user, but also may have to provide for the demands of multiple users, for example giving them access to the same file.

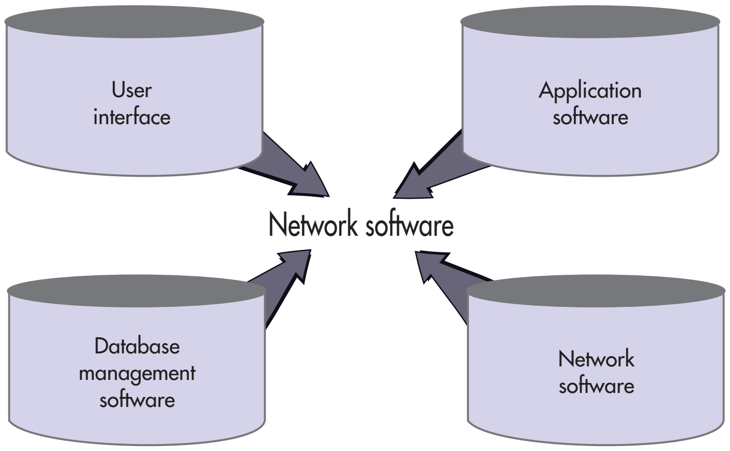


Figure 2.21 A client/server software solution involves four parts.

Customised off-the-shelf packages

The rapid increase in the use of personal computers has been accompanied by a similar increase in the range and type of off-the-shelf software packages. These packages have been designed to perform general tasks such as word processing and database management. Some of these applications, such as word processors, will perform the most common tasks without the need for modification. Other applications will need to be modified on installation in order to be able to perform their tasks. For example, a bookkeeping program will need to be set up to reflect the way in which a business manages its affairs.

A large number of current applications allow the user some form of choice in the manner in which the program operates. This choice may be as simple as a game player being able to choose the keys used for control through to the customisation of the complete user interface.

Developers can benefit from this type of software as it provides a quick solution to a large number of problems. Tasks performed in implementing one of these solutions will include a customisation of the interface, production of appropriate reports and conversion of data from the old system. This type of software generally lends itself to a prototyping approach or an end-user development approach.

The popularity of customisable software is due to a number of factors. The cost of development is reduced, as the manufacturer has done a large amount of the process development. Development time is also reduced by the use of customisable software. The producer of the application package often provides technical support to purchasers.

Figure 2.22 A small business, such as a video store, will often use customised software in preference to purpose-built software.

Exercise 2.4

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Developers can obtain a quick __________ to a large problem by using __________ packages that can be __________ to reflect the way the business operates. This software is used by __________ the user interface, producing appropriate __________ and __________ the data from the __________ . Both the __________ and __________ of development are reduced as the __________ has developed a lot of the __________ . A further advantage is often the __________ provided by the producer of the package.

2 Name the four sections of client/server software. Describe the responsibilities of each of these sections.

3 Explain how the evolution of computer technology has helped change the way in which software is developed.

4 As the technology has changed, so have the programming environments that developers work with. Explain why languages such as Visual Basic and Java would not have been possible in the 1960s and 1970s.

5 Explain the meaning of the term 'outsourcing'. In what ways has the move to outsourcing software development affected the employment of computer professionals?

6 In this section, we have investigated a number of trends in software development. Investigate other trends that have not been mentioned. Present your findings as a fully edited report on both disk and paper.

Use of CASE tools and their application in large systems development

Computer-assisted-software-engineering (CASE) tools have been developed to help software developers in the tasks of software development and maintenance. Some of these tools have been designed to assist with a particular activity or function in software development. Other tools are combined to form an integrated development environment. An integrated set of tools is often referred to by the acronym ICASE.

Individual CASE tools will be geared to support activities such as problem definition, planning, building, testing and modification. In this case the tool will perform a number of tasks that contribute to the completion of a particular stage in software development. Other tools will be oriented towards particular functions within the software development cycle. In this case tools will perform a particular task; for example, tools are available for text editing, test-data generation and document preparation. Commercial software such as word processors and drawing applications may also be used to assist in the development of software. For example, a number of application programs contain flowchart construction modules that can be used in the development process.

Software versions

As software is progressively revised, it becomes harder to keep track of the versions in use. This is especially true for large systems, which have a number of different software installations. A number of CASE tools are available to help with this task.

A problem with continually evolving software is the management of the different versions. In order to help with this process, each version is given a release name or number. The most common method is to use a numbering system similar to the following. The first version is labelled as version 1.0. Using this numbering system, major version changes are given numbers 2.0, 3.0 and so on, with updates being given version numbers such as 1.1, 1.2, 1.3. This numbering implies that version 1.2 has been derived from version 1.1 and 1.3 has been derived from 1.2, but this is not necessarily the case. Version 1.2 may have been derived from version 1.0, but with a different set of modifications than those applied to create version 1.1. For example, version 1.2 may have been developed to test a modified interface, with no intention to release it for use. Thus the release versions might be 1.0, 1.1 and 1.3, then version 2. Bug fixes and other small changes may be numbered by adding a further digit. For example, a bug fix on version 1.1 may be numbered 1.1.1 as it presents no apparent change to the user of version 1.1.

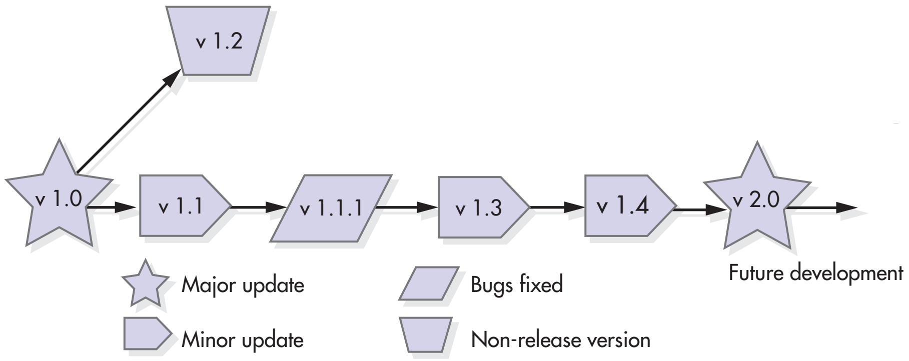


Figure 2.23 Versions are numbered to keep track of the changes.

The management of the changes to a software product is known as **software configuration management (SCM)**. To avoid conflicting changes being made by different parts of the development team, all proposed changes need to pass through a central body so they can be evaluated and tracked. It is also very important for the changes to be properly documented for the user, as they may involve some change to operating procedure.

A number of CASE tools are available to track the changes made to a software solution. Some of these, for example *AIX CMVC*, *CMF*, *Continuus* and *PVCS* are specific management solutions. Some integrated CASE tools, such as *AIX SDE WorkBench* and *SPARCworks Toolset*, contain version management components.

Data dictionary

As discussed in the Preliminary Course, a data dictionary is essential to the software development process. This is especially true for projects that are being created by a large development team. The use of an appropriate networked CASE tool to manage the data dictionary will enable all developers to have an up-to-date data dictionary. The use of a CASE data dictionary tool also assists in the management of large projects where the data dictionary is complex and therefore almost impossible to manage manually. Sophisticated integrated CASE tools are also able to create a data dictionary from a dataflow diagram that was produced within the CASE environment.

Figure 2.24 CASE tools help in the management of collaborative programming.

Data dictionaries produced in this manner will contain a number of entries for each data item. Common entries for each data item will generally include:

- *identifier:* the main name of the control or data item, for example *cost*
- *alias:* alternative names for the same data item; for example, if swapping is to take place, a temporary data item, such as *cost_temp*, will be required
- *processes:* the processes that use the data item together with the way in which the data item is used; in our example *cost* may be calculated as being *quantity* multiplied by *unit_price*
- *other information:* a generalised category which includes data type information, restrictions or limitations on the data item and any pre-set values.

Test data

Test-data sets fall into two categories: data to test the correct workings of modules, and test data to evaluate the performance of the system under simulated working conditions. There are test-data generators whose purpose it is to create large sets of test data. These test-data generators can be given the data specifications and syntax for a particular data set and they will then generate large amounts of data that can be used to evaluate the performance of the system. The outputs from the system still need to be checked, but some of this checking, for example checking for particular error messages associated with certain data, can be performed automatically.

A program known as a file comparator can be used to compare the outputs of two different versions of the same program. The same input data is entered into the two versions, the output being saved. The file comparator can then scan the output files and indicate where and what the differences are.

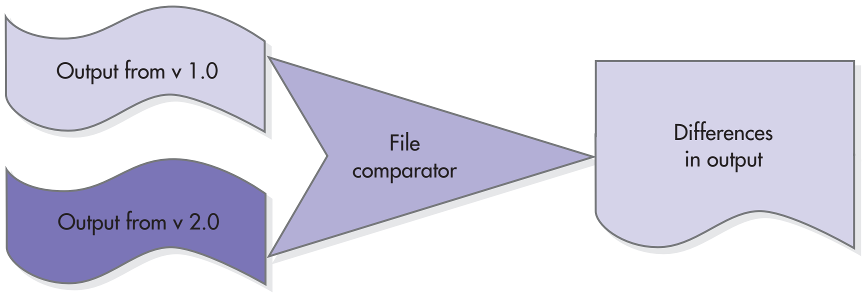
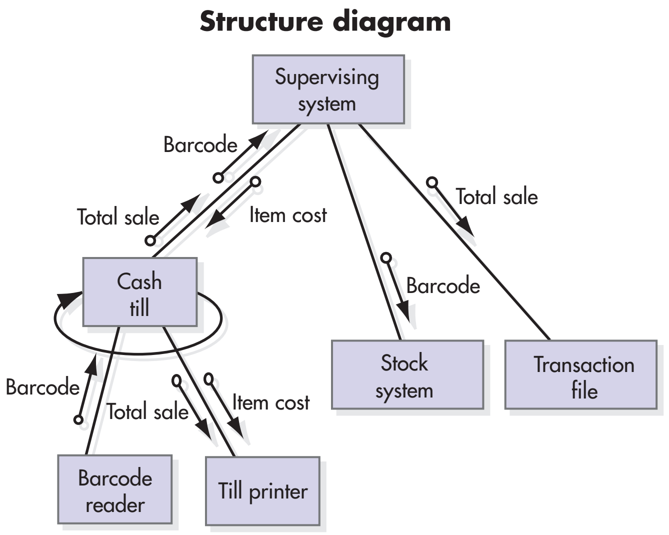


Figure 2.25 A file comparator compares two files for differences.

Production of documentation

Throughout the software development cycle the production of documentation is a continuous process. Programmers often make use of existing applications to assist with this process. For example, dataflow diagrams may be easily produced using a draw program. It makes sense, therefore, to incorporate the production of documentation within a CASE environment. Allowing the developer to produce documentation at the same time as the development process is taking place may also avoid the delay in its production. A developer who has access to the documentation tool from the same workstation as development is taking place will be more likely to use it. A number of different CASE tools are available to assist with the documentation process.

A graphical application can be used to create systems representations such as dataflow diagrams, system flowcharts and structure charts. These different representations use graphical symbols to represent different aspects of the system. Many of these symbols are common to two or more of these representations. By giving the programmer access to symbol templates, the graphics program can speed up the process of creation. A further benefit from this approach is that it enables changes to be made to the representation without the need to completely redraw the diagram.



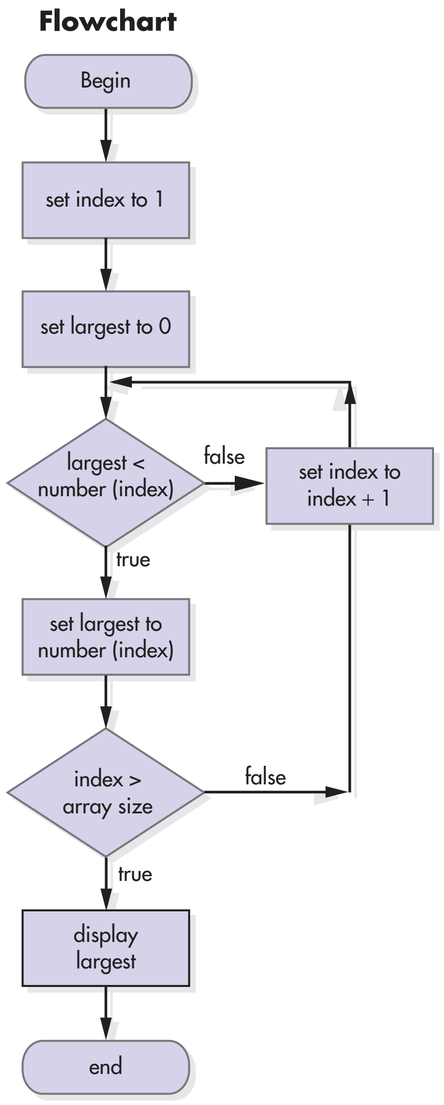


Figure 2.26 Different representations of systems often use the same symbols.

Graphics based CASE tools include:

- *Adobe Illustrator:* can be used as a general drawing and text-handling tool
- *Create:* a visual software development program that creates flowcharts
- *Gsharp:* a graphics tool for SUN, DEC, IBM and other mainframes and minicomputers
- *Rose*/C++: a graphical tool that can be used for analysis, design and implementation in C++
- *Smart chart:* a tool used to automatically generate structure charts.

A number of different text-based documentation tools are available within different CASE environments. The tasks performed by these tools vary from project management through to the production of code in a supported language. As already mentioned, the humble word processor can be used to produce various forms of documentation. A word processor can also be used to create an ASCII text file that can be compiled.

A number of document-oriented CASE tools exist. Their purposes vary from templates through to automatic document producers. Some of the tools available at the time of writing are:

- *ALDA:* a tool that is used to create and use advanced technical documentation
- *DocBuilder:* a tool that automatically produces various types of documentation including specification documents, design documents, and quality and test coverage reports
- *LifeCDM:* a document management system that covers publishing, management and distribution of documentation.

Exercise 2.5

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

CASE tools have been developed to ________ software developers. These tools will either help with a ________ or perform a ________ which is a part of the ________ development process. In a large development ________ a CASE tool may be used to create a ________ which lists all the data ________ together with various details. CASE tools are also used to ________ the changes in software. This is important as several different ________ of a program may be in use at the ________ time.

2 Examine the application software you have loaded on your school computers and list the software development tasks that you can perform with each of the applications.

3 Explain why CASE tools may be used in the development of a large system.

4 Explain why it is important to be able to identify the different versions of a computer program. Use an example to illustrate your answer.

5 Use a graphics program to create templates for all the symbols that you use in your course. Keep this template file and update it each time you meet a new graphical technique. To start with, your template should contain all the flowchart symbols and the railroad diagram symbols. You might also like to include common modules such as pre-test and post-test loops.

6 Use an appropriate application to create a template document that you can use for a data dictionary in all your programming activities.

7 Investigate the availability of CASE tools on the Internet. At the time of writing, a good site to start at is http//www.cern.ch/PTTOOL/SoftKnow.html. You can then use this information to follow up on particular tools through the vendor's sites. Report on your findings, including as references any Internet sites you used.

Team Activity

Create an appropriate website for 'Bikes To Go' by using a prototyping approach. Evaluate the prototype and make suggestions for its improvement. Do not forget to provide the proper documentation. Each member of the team should evaluate the prototype, one member as management, one member as an employee and at least one member as a customer.

Review exercises

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

A large program is most likely to be created using a ________ approach. The advantage of this approach is that a ________ of this size is created by a ________ . Members of the ________ need to be kept up to ________ with changes. The formal nature of the ________ encourages the production of ________ at each stage of ________ . CASE ________ can also be used to help with the ________ as each member of the team can access the latest ________ to the program.

2 Choose the alternative, A, B, C or D, that best answers the question.

a A publisher wants to put its textbooks on CD with links to the Internet. The best development method for this project would be:
- A end-user development
- B a prototyping approach
- C rapid application development
- D a structured approach

b A library installs and implements the catalogue module of its new management program while the other functions are being performed by the old program. This is an example of:
- A direct cut-over
- B parallel conversion
- C phased conversion
- D pilot conversion

c The interface of a client/server application must:
- A be responsible for processing inquiries
- B meet the software requirements of the user
- C communicate the state of the application and network messages
- D provide the means for network communication to take place

d A word processor can be used as a CASE tool. Which one of the following tasks cannot be undertaken by a word processor?
- A checking the syntax of a coded algorithm
- B coding an algorithm in a language
- C creation of the user manual
- D production of a data dictionary

e The issue of access to data in a new system is an example of:
- A constraints on the development
- B operational feasibility
- C social and ethical considerations
- D technical feasibility

3 Describe the roles that the systems analysts, programmers, management and users play in each of the stages of the structured approach.

4 A new computer program has been developed to manage an aircraft in flight. Which of the four implementation methods would you use for this application? Give reasons for your choice.

5 Use the classified advertisements in a newspaper to investigate the types of employment being offered in computers and information technology. Report on the conditions of employment (for example whether the job is full-time).

6 What problems do developers of network software have to overcome that developers of single-user software do not?

7 A sheep stud keeps its breeding records in a number of books. The owners are now finding it difficult to keep track of the data and have decided to develop a program to perform this task. Choose an appropriate development approach and justify your choice. What CASE tools, if any, would you use to help with this task?

Chapter summary

- The four main approaches to software development are the structured approach, prototyping, rapid application development and end-user development.
- The approach taken depends on the needs of the project and skills of the development team.
- The structured approach is the most formal and consists of five stages: defining the problem, planning the solution, building the solution, checking the solution and modifying the solution.
- The problem-definition stage involves the systems analyst who examines the existing system and determines the requirements of the new one.
- Management and users are also involved in the problem-definition stage.
- The feasibility of a solution is also examined at the problem-definition stage.
- The planning stage involves choosing the data structures and program structure, designing the algorithms, planning the user interface, scheduling the project and choosing a programming language.
- The planning stage involves the systems analysts and the programmers.
- The building stage involves the programmers who code the algorithms and test the solution.
- The checking stage involves testing the program with real data to ensure it works properly.
- The checking stage often involves the users and is called acceptance testing if employed on custom software and beta testing if employed on application software development.
- Prototyping involves building a working model that the users evaluate. The prototype is then modified and evaluated further.
- There are two types of prototyping: the first is used to gather information; the second is used to evolve a fully working solution.
- Rapid application development (RAD) involves the use of existing modules to create a solution.
- RAD uses automated tools to help with construction.
- End-user development is usually for small, personal solutions. They are usually created with the help of an application program.
- A combination of approaches may be needed to solve a problem.
- There are four methods of conversion from the old system to the new: direct cut-over, parallel conversion, phased conversion and pilot conversion.
- A direct cut-over involves a complete and immediate change to the new system.
- Parallel conversion involves the old and the new systems being fully used together.
- Phased conversion involves the gradual implementation of the new system.
- In a pilot conversion the new system is fully installed but only some of the functions are used, the old system still being used for the other functions.
- The change in the availability of technology has changed the way software is obtained.
- The contracting of an outside company to develop software is known as outsourcing.
- Big businesses often use a client/server approach to software development as they employ networks of personal computers.
- Individuals often customise off-the-shelf software to solve their problems.
- Popular languages are easy to use for people with little or no formal computer training.
- Computer personnel are employed more often on a contract basis than a full-time basis.
- Off-the-shelf software packages can be customised to provide a quick solution to a problem.
- Computer-assisted-software-engineering (CASE) tools can help with the development of a large system.
- CASE tools help in tasks such as tracking software versions, creating and maintaining data dictionaries, the creation of test data and the production of documentation.

chapter 3

Defining and understanding the problem

Outcomes

A student:

- differentiates between various methods used to construct software solutions (H 1.2)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- constructs software solutions that address legal, social and ethical issues (H 3.2)
- identifies needs to which software solutions are appropriate (H 4.1)
- applies appropriate development methods to solve software problems (H 4.2)
- applies a modular approach to implement well-structured software solutions and evaluates the effectiveness of the solutions (H 4.3)
- applies project management techniques to maximise the productivity of the software development (H 5.1)
- creates and justifies the need for the various types of documentation required for a software solution (H 5.2)
- selects and applies appropriate software to facilitate the design and development of software solutions (H 5.3)
- assesses the relationship between the roles of people involved in the software development cycle (H 6.1)
- communicates the processes involved in a software solution to an inexperienced user (H 6.2)
- uses a collaborative approach during the software development cycle (H 6.3)
- develops effective user interfaces, in consultation with appropriate people (H 6.4)

Students learn about:

Defining the problem

- identifying the problem
  - needs
  - objectives
  - boundaries
- determining the feasibility of the solution
  - is the problem worth solving?
  - constraints
  - budgetary
  - operational
  - technical
  - scheduling
  - possible alternatives
  - social and ethical considerations

Design specifications

- the developer's perspective in consideration of:
  - data types
  - algorithms
  - variables
- the user's perspective

Modelling

- representing a system using diagrams, including:
  - input, process, output (IPO) diagrams
  - storyboards
  - data flow diagrams
  - systems flowcharts
  - screen designs
  - consideration of use of a limited prototype

Communication issues, including:

- the need to empower the user
- the need to acknowledge the user's perspective
- enabling and accepting feedback

Students learn to:

- develop and interpret design specifications from a user's perspective, considering:
  - screen design
  - appropriate messages
  - appropriate icons
  - relevant data formats for display
  - ergonomic issues
  - relevance to the user's environment and computer configuration
  - social and ethical issues
- evaluate the extent to which a proposed system will meet a user's needs
- differentiate between the different forms of systems documentation and the purposes for which each is intended
- interpret a system presented in a diagrammatic form
- create a diagrammatic representation for a system using an appropriate method
- effectively communicate with users regarding a proposed software solution

Defining the problem

In the Preliminary Course we examined the three stages in problem solving: understanding the problem, working out the solution and checking the solution. There are a number of aspects involved in understanding a problem.

Identifying the problem

Problem definition involves examining a number of factors, including the needs of users, the objectives that the solution is to meet and the boundaries within which the solution has to operate. A further consideration, known as **feasibility**, is whether it is actually possible to implement a proposed solution.

A number of different techniques are needed to ensure that all the factors that contribute to the problem are identified. The use of these techniques will guarantee that all contributing factors have been identified, and will ensure that the solution meets the requirements of the end user.

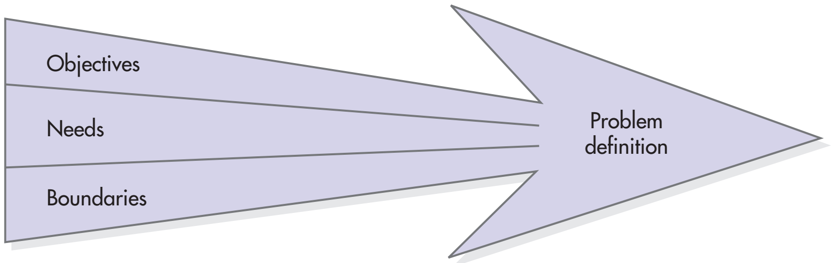


Figure 3.1 The needs, objectives and boundaries of a system contribute to the definition of a problem.

Needs

When defining the needs of a system, we need to focus on more than the needs of the user. The user's needs are an important factor, but there are other considerations. The needs of the system will also involve the need to represent various data items, the need to store certain facts, the need to perform certain processes and the need to output data in a particular manner. For example, the needs of a system to handle the bookings of an airline are entirely different from the needs of a system designed to help navigate an aircraft. One of the needs of the booking system is to be able to store large amounts of data that can be accessed from any one of a large number of terminals; the navigation system will be accessed primarily from one location. There can be a delay of some seconds between making an enquiry and obtaining the answer from a booking system; a similar delay in using a navigation system could prove disastrous.

Human needs are probably the easiest for systems analysts to identify since they can mentally put themselves in the position of the user. Activities such as surveys, interviews and observation identify how people interact with the current system and highlight the problems they face. It is important to determine the needs of all classes of user, whether direct or indirect, so that the new system can perform properly.

Figure 3.2 The needs of all types of user should be established.

Other system needs are determined by the type and amount of processing required. For example, graphics systems to process full-motion video images in real time need a very fast processor and large amounts of primary storage. Systems such as personal organisers can be created with much slower processors and smaller primary storage since the data being processed is much less complex. The manner in which processed data is to be presented will also determine the needs of the system. A system that is to output dictated text to paper would have differing needs from one that reads handwriting and outputs it as speech.

Figure 3.3 The desire to manipulate sounds using a computer brings with it a number of special needs.

Objectives

The needs of a system can be analysed to determine the way in which that system must perform. The criteria that the performance is measured against are known as the objectives of that system. For example, an aircraft navigation system will need to supply the various outputs in **real time** (that is, as a value changes, the change must be reported immediately). So one of the objectives of this system may be to calculate the altitude every fiftieth of a second and display it. Expressing the objectives in a measurable way provides a means by which the program can be evaluated.

The objectives of a system can be placed into categories. Some objectives will relate to the interaction with the user, such as a maximum response time. Other objectives might describe a minimum number of transactions that can take place within a particular period. Other objectives might specify how the output is to be presented; for example, an objective of a real time video image processing system would be that the vision and sound will be output simultaneously.

In determining the objectives, the needs of the system are examined and converted into a number of statements. These statements are written in such a way that they provide a means of evaluating the performance of the new system. For example, a manufacturing company may wish to automate the production line. The need of the company is to perform several of the manufacturing tasks using robots. This need is then translated into certain objectives, such as the production line being able to produce a certain number of articles within a particular time.

Figure 3.4 An automated production line has to meet a number of objectives to be successful.

Boundaries

A system performs a certain function within a wider environment. For example, your nervous system carries messages between the various parts of your body. Your body is the environment within which the nervous system works.

Each system interacts with one or more other systems that exist within the same environment. The outputs of one system become the inputs of the next one in the chain. For example, the railway network can be seen as an environment that consists of three systems: one to accept passengers into the network, one to transport the passengers and one to send passengers out of the network. Within an environment, the places where outputs leave one system and pass to the next are called **boundaries**.

Although it might appear to be trivial, it is important to identify where the boundaries of a system are to be. Different people will have different ideas as to where a boundary between two systems is. In the railway example, the boundary between the transport system and the system that passes passengers out of the network could be seen as being the platform edge or it could be seen as being the ticket collector. When we develop a system, we need to ensure that the boundaries are clearly identified. In a team approach to software development, each team member might have a different idea as to the boundaries of the system. If this were allowed to continue through development, it is possible that the program would not function properly. The chosen boundaries can also influence the way in which the system outputs will be presented.

Figure 3.5 We can think of the platform as the boundary between two systems of the railway network.

Determining the feasibility of the solution

There is a great difference between wanting something and actually achieving that goal. For example, Janine needs to go to London. She would really like to fly in her own aeroplane, but she knows that there is no way she can afford it. So her problem cannot be solved by this means. In other words, that solution to her problem is not feasible. However, other solutions, such as flying with an airline, can be implemented.

The feasibility of a solution will depend on a number of factors. The factors range from the obvious ones, such as whether the problem is worth solving. through to those that are less obvious, such as whether the solution can be implemented within a reasonable time frame. When a system is developed, these factors are examined and reported on in a **feasibility study**.

Some criteria that contribute towards the feasibility of a project can be thought of in terms of more than one factor. For example, if a particular hardware item needs to be developed and the development will be expensive and time-consuming, the need for the item will contribute to the technical feasibility, financial feasibility and schedule feasibility.

Is it worth solving?

The most obvious of the factors that need to be considered when determining the feasibility of a solution is whether the problem is really worth solving. If the problem is a trivial one, it is probably not worth spending the time and effort on devising a solution. For example, if you don't like the colour

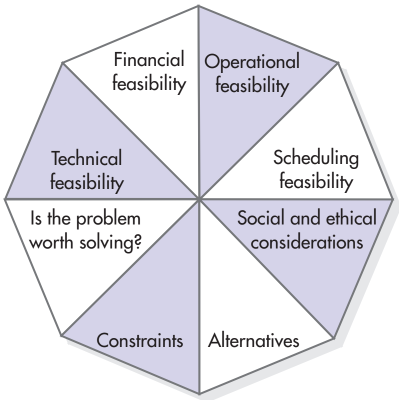


Figure 3.6 A number of different factors will determine whether a solution is feasible.

of the car that your parents have just bought you, paying for a complete re-spray of the car is not really a sensible solution. Changing the colour of the car would not affect the way in which the car performs its task. However, if the brakes on the car needed attention, you would have no hesitation in having them repaired, as not solving that problem would be very dangerous.

Constraints

A **constraint** is a limitation that is placed on something. Constraints are many and varied, but all will have some effect on a project. A constraint may be as obvious as a limited budget, or it may be the inability of staff members to accept a change. The nature of all the constraints must be fully investigated, as the design of the new system will have to take place within these constraints

As the design process progresses, some of the original constraints may be eased and others may appear. For example, if we were going to design a new television set, we might like to incorporate a satellite-receiving dish within the set. At the moment this technology is not available, so the design is constrained by this fact. If, however, a new technology appears that allows the dish to be incorporated within the television, the constraint has been removed. As computer technology is advancing very rapidly, what begins as a technical constraint for a software development project may not be one by the time the project has been completed.

Budgetary considerations

Budgetary considerations will come from a number of sources. The development cost of a system is only one of the costs that has to be met. Other costs will include the initial cost of implementation and the ongoing running costs. A business must be able to afford to implement a system in order to benefit from its implementation. However, the additional income and/or the projected savings need to be more than the cost of implementing the solution.

The **cost–benefit analysis** is an important part of the budgetary component of a feasibility study. The purpose of a cost–benefit analysis is to weigh up all the costs involved in implementing and operating the new system. The costs of the system, together with the running costs, are calculated as accurately as possible over an estimated lifetime of the system. These costs are then weighed up against the increases in income and decreases in costs that will flow from the new system. The calculations are incorporated in a detailed report that also provides an opinion as to whether the project is financially worthwhile.

Operational considerations

The purpose of a new computer system is to meet the needs of the organisation. In meeting these needs, the new system will have some form of impact on the members of the organisation, from management through to the direct users.

The main way in which the new system will impact on these people is in the area of manual processing. In order to be successful, the computer system must be complemented by an efficient manual system. If the requirements for manual processing are too difficult to implement, the overall performance of the system will suffer. A number of factors affect the ability of an organisation to implement new manual processes.

Tasks carried out by personnel within the organisation are usually very clearly defined. If the implementation of a new system changes the nature of this work, the issue of redefining the worker's role becomes important. In some cases this can be done simply by negotiation with the individual. In other cases negotiation with union personnel may be required. Further problems in defining new roles may be caused by government regulation.

When tasks within an organisation are redefined, some job descriptions might cross old boundaries. For example, the introduction of garbage trucks with bin-lifting mechanisms meant that the driver became responsible for emptying the bins as well as driving the truck.

The crossing of boundaries can often redefine the jobs of those workers who are not directly involved with the system; for example, middle managers might find that their department has been incorporated into one or more other departments. This effect can force the organisation into a complete restructure.

All of these factors need thorough investigation before the creation of the new system, as some operational factors may prevent the successful implementation of the system.

Figure 3.7 Changes in work practices may cause resentment or prevent the implementation of a new system.

Technical considerations

The technical feasibility of a proposed solution is concerned with establishing whether the hardware and software needs of that solution can be realistically met. Users who are unfamiliar with the technical aspects of computers can have totally unrealistic concepts of the tasks that can be performed by them. For example, English teachers might like to have a program that marked English essays. The computer technology exists that can input handwritten text for processing, but software that can accurately mark a student's essay would be extremely difficult, if not impossible, to create.

Since the success of the project depends on hardware and software being available to perform the required tasks, the systems analyst needs to have a sound knowledge of the latest developments in the field as well as extensive programming knowledge.

As with the other areas of a feasibility study, the technical feasibility study must be thorough and accurate. If this area is not properly investigated, a technical hitch could occur halfway through the project which could lead to the project being abandoned.

Scheduling

There is often a need to implement a new system within a fixed time limit. The best example of this was the need to overcome the 'Y2K' problem before the year 2000 began. The limited time for implementation also places pressure on the development team to complete all activities before the deadline. Thus scheduling feasibility will not only investigate the feasibility of a project completion date but will also report on the most desirable development approach. When there is limited time before implementation, the best option for implementation may be a modification of the existing system rather than the development of a completely new one.

Possible alternatives

Once all the factors have been investigated, a number of alternative solutions, including leaving the system unchanged, are proposed. Included in each proposal is a rough estimate of the cost of the solution. The advantages and disadvantages of each of the proposals are documented. Finally, a recommendation is made by the analyst and passed on to management for a decision.

Social and ethical considerations

The introduction of a new system will always have an impact on people. Some of the impact will be felt by management, some by direct users such as employees and some by indirect users such as customers. These impacts can be thought of as social impacts and ethical impacts.

Previously it was noted that there is often a change in the work practices that accompany the introduction of a new system. This is an example of one of the ways in which the system will impact socially. The change in work practices may mean that some employees lose their jobs, while others are required to perform new tasks. Some of these effects may spread outside the organisation. For example, employees who lose their jobs will need to find other jobs.

Figure 3.8 The introduction of ticketing machines in Melbourne had a wide-ranging effect on the population.

Sometimes the introduction of a new system will impact on the general population in quite a significant way. The introduction of automatic ticketing machines on Melbourne's public transport system affected not only the employees but the public who used the system. A number of social problems emerged, ranging from trouble with access for the disabled through to an increase in fare evasion and vandalism.

Ethical considerations also need to be taken into account when introducing a new system. This is especially important when private or sensitive personal information is involved. Each individual has a right to privacy and steps should be taken to ensure that, with the introduction of the new system, this right is respected. Also, if any code or programs from outside sources are to be used, copyright issues need to be addressed.

Exercise 3.1

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

The three stages of problem solving are __________ the problem, __________ the solution, and __________ the solution. Defining the problem involves examining the __________ of the users, the __________ that the solution has to meet and the __________ within which the solution has to operate. The other consideration is __________, that is, whether it is actually __________ to implement a proposed solution. All __________ that contribute to the problem must be __________ so that the solution meets the __________ of the end __________.

2 Apart from the needs of the user, other needs must be met when designing a new system. Name and describe these needs in your own words.

3 As well as needs, each system must meet certain objectives. What is meant by the term 'objectives'? Describe, using examples, the types of objective that have to be met by the new system.

4 Explain what is meant by the term 'boundary' when applied to a system. Use an example to illustrate your answer.

5 What is a feasibility study? What is the purpose of a feasibility study in developing a new system? Use an example to illustrate your answer.

6 Describe the different factors that need to be taken into account when investigating the feasibility of a solution.

7 A school wants to improve its school report system. Briefly describe the needs of such a system.

8 For the school report system, describe some of the objectives that it will have to meet. Include some performance objectives and some objectives relating to the user interface.

9 Describe some of the factors that should be taken into consideration when determining the feasibility of the school reporting system.

10 What are the social and ethical considerations that have to be taken into account when designing the school reporting system?

Design specifications

Once a system has been chosen from the options, the design process has to be considered. This process should not be rushed. Careful planning of the development of the system will ensure that development resources are used efficiently. The design specifications will lay down the guidelines for development and give a yardstick that can be used to measure the success of the project. The specifications cover a number of different aspects of the design process: the scope of the project, data design, overall program design, interface design, process design, a cross-reference with the requirements definition and implementation and testing of the final project.

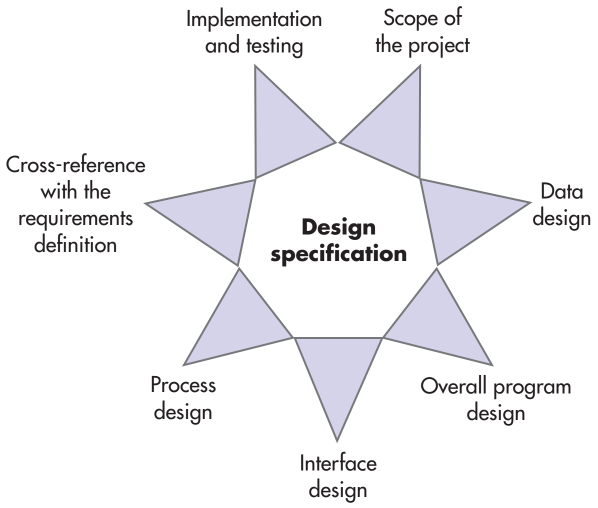


Figure 3.9 A design specification covers a number of different aspects of the development process

The *scope* of the software project provides detail about how the software fits in with the whole project. The scope also defines the boundaries within which the software will operate. Also included are the forms of the input and output data, together with a description of the processing needed to convert the inputs to outputs. Special processing needs are also included within the scope statement.

The *data design* section of the design specification will identify the data objects required for processing and the resulting data types required for both internal and external use. File structure and access are also covered in this section. The final task of the section is to cross-reference the data items to their appropriate files.

In the *overall design of the program*, the system is represented by a series of **structure charts** (see p. 66 for a full description) which represent program modules and the interactions between them. These are used to illustrate how the program will be constructed.

The *interface design* specifications are concerned with the human–machine interface and the interface between modules. This involves specifying how the user will interact with the program, together with a set of design rules for the interface. Equally important to the design of the program are the specifications that describe the interfaces between modules and between the program and external devices.

Each module within the program is then described in detail. For each of the modules, a description of the *processing* is written. Also included within this section is a description of the interface, the algorithm or algorithms that describe the module, the local data structures used within the module, and any restrictions or limitations that may be placed on the processing.

Cross-referencing the requirements definition to the software design process ensures that all the user's requirements will be met by the software. The cross-referencing also indicates which of the modules are important in meeting each of the particular requirements.

The processes of *implementation and testing* need to be designed in the same way as any other process. It is not just a matter of testing each module, combining the modules and testing further, then putting the system into place and hoping for the best. As with any process, implementation and testing procedures need to be specified before they are used. In this section of the design process the general processes for testing are outlined. As each module is developed, it will be tested within the general guidelines, although special testing methods may also be required. If special methods are needed, they will again be designed before the construction of the module takes place. It is also important that the installation onto the customer's site is carefully planned, as the installation of the system will disrupt normal working practices.

The developer's perspective

A developer views the solution of the customer's problem differently from the way the customer views it. The customer wants the problem solved, and has little interest in how the solution is arrived at. The developer is faced with the task of converting the customer's wishes into a computer program. Thus, the developer must look at the problem in terms of data types, variables and processes.

Data types

The choice of suitable data types for use within a program is one of the most important decisions a programmer has to make. Without appropriate data types, algorithms can be difficult, or even impossible, to design. A well-chosen type can make an algorithm much simpler to design and implement.

Three factors influence whether a particular data type is appropriate for the task. The first is whether the type is able to cater for the storage of all possible data items. The second is whether the type can be visualised by the programmer and expressed as a model on paper. Finally, a data type has to reflect the data and be able to allow the sort of processing required by the problem. These examples illustrate how to decide on an appropriate data type.

Example 1

PROBLEM

A program is to be written to monitor the takings of a cash register. The program will only output the total day's takings at the close of business.

Figure 3.10 The takings of a cash register can be monitored by a program.

DATA TYPE

The data is quite clearly numerical, so the use of character, Boolean or string data types is not appropriate. Since the transactions do not have to be stored individually, an array is not required. (If the problem specification had required that each transaction needed to be stored, an array would have been needed.) This leaves us with a choice between the integer and real number types. Some may consider using an integer data type with all amounts being expressed in cents, but since the maximum integer size is usually 32767 ($2^{16} - 1$), the program may not be able to cope with some days' takings (as 32767 cents is only \$327.67). The more obvious reason for choosing a real number data type is that an amount of money is normally expressed as a floating point number. This means that we would use real variables for the amounts.

Example 2

PROBLEM

When an item's barcode is scanned at a supermarket checkout, the product code is checked to determine whether the scanning process is successful. The check involves adding the sum of the digits in the odd positions to three times the sum of the digits in the even positions. If the result of the addition is divisible by ten, the object is deemed to have been successfully scanned. A data type has to be designed for a barcode verification module. For example, in the barcode 9 7 8 0 8 5 8 5 9 6 3 7 5 the digits in the odd positions (that is 9, 8, 8, 8, 9, 3 and 5) add up to 50 and the digits in the even positions (7, 0, 5, 5, 6 and 7) add up to 30. When the values 50 and 3 × 30 are added, it produces a sum of 140 which is exactly divisible by 10, so the barcode is valid.

Figure 3.11 A scanned barcode needs to be verified before it is used.

DATA TYPE

An initial look at the problem suggests that a numerical data type may be suitable. However, this is not the case, as extracting each of the digits from an integer or a real number requires a complex algorithm and may take excessive processing time. This is not the only reason for rejecting these data types. There is also a problem with the internal representation of both integers and real numbers when applied to this problem. Most barcodes have 13 digits, which means that the normal two-byte integer data type is not capable of handling values that size (remember that the range is –32768 to +32767). In fact, six bytes would be needed to give the required range in twos complement form. As discussed in the Preliminary Course, the use of floating point representation involves approximating values, so the barcodes may not be exactly representable in binary. A more suitable data type would be an array of characters. By choosing an array of characters, each digit becomes individually accessible, thus simplifying the algorithm. The storage required (13 bytes plus one or two extra bytes to store the array size) is only twice that of integer representation but has the added advantage of easier processing.

Example 3

PROBLEM

Shape recognition is used in many computer applications, one of them being OCR (optical character recognition). An algorithm is required for a computer program that reads a special set of printed characters from a cheque, turning them into numerical data for further processing. The 'character scanner' contains a lens that projects the image of the character being 'read' onto a grid of light-sensitive cells. An algorithm is required which matches a pattern of light on the grid to a particular character.

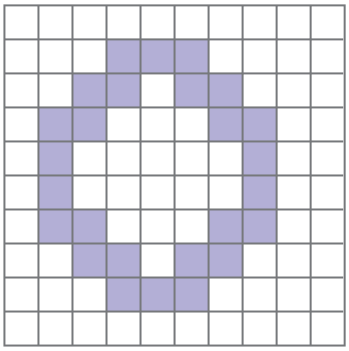


Figure 3.12 A shadow pattern on a sensing grid can be represented by an array.

DATA TYPE

Each of the cells in the grid responds to light by creating an electrical impulse which is digitised by a computer interface. In this way the information being interpreted by the computer program is a series of 0s (corresponding to dark patches) and 1s (corresponding to light patches). On paper the detector can be represented by a two-dimensional grid, and this representation can also be used for the data type. As data enters the recognition system as a series of 0s and 1s, we only need a data type that can represent two states; this leads to the choice of the Boolean data type. Consequently, the most suitable data type for this application is a two-dimensional array of Boolean data, with its dimensions equal to the number of rows and columns of the sensing grid (see Figure 3.12).

Example 4

PROBLEM

A second module is required for the barcode scanning and pricing process mentioned in Example 2. The purpose of this module is to match the scanned barcode to the items in a database and return the item's description and price.

DATA TYPE

The data required to be stored for each item in the supermarket is of varying types. The description of each item is a string of characters, the price is a real number and, as decided above, the barcode is to be stored as a one-dimensional array of characters. Since each of these data elements needs to be stored under the one broad description, a record is needed for each item stocked by the supermarket. However, the supermarket stocks more than one item, each needing the same elements, so an array of records is required.

Variables

Variables are used by programmers to represent storage locations within the computer. A programmer will look at the entities (data items) that need processing and determine how they are going to be accessed. Part of this process is to determine the data type that is to be used to represent the data item; the other part is to choose an appropriate identifier (name) for the data item.

The choice of an appropriate identifier is very important in the process of programming. If identifiers are not given names that represent the item being processed, then it becomes difficult to follow the logic of the program without making notes. (These notes are usually included as comments or remarks.) This choice of names is known as internal documentation.

For example, if the variable identifier *X* was chosen to represent the amount of GST charged on a service, another member of the development team who had to read the algorithm would have no indication of what was being processed. It is much better to use a name such as *GST_value*. (Note that the underscore has to be used since spaces are used to separate words.)

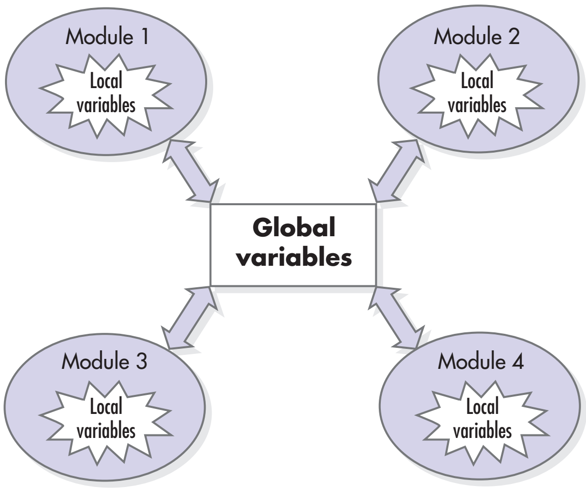


Figure 3.13 Global variables are accessible to all modules. Local variables are accessible only to the modules for which they are defined.

The use of variables within the program is also of importance. The programmer must make sure that data values are made accessible to the modules that need them. This means that the programmer has to look at the use of global variables to represent these items. Other variables will only be needed within a module, so they can be created as local variables for that module. The distinction between these types of variable is made so that unpredictable changes are not made to the values of variables.

Algorithms

In order to construct a successful solution to the problem, the programmer has to understand the processes that need to be carried out by the program. This understanding then has to be translated into the algorithms that will form the program. The programmer's experience, together with various systems descriptions, will be used to design the algorithms required by the solution.

As seen earlier in the course, few programs consist of a single simple algorithm. This means that the programmer has to break down the problem into smaller modules until each of the modules can be expressed as a simple algorithm. The process is known as **top-down decomposition**. While decomposing the problem into these smaller units, the programmer must continually ensure that the client's problems are being addressed rather than trying to make the program elaborate.

Once each of the algorithms has been identified, the programmer will look for familiar modules. The familiar modules can often be drawn from the programmer's library, modified for the new application and inserted with minimal testing. For example, most programmers have a regular module for accessing each of the elements of an array. This module can be modified for any process that requires each of the array elements to be accessed in turn.

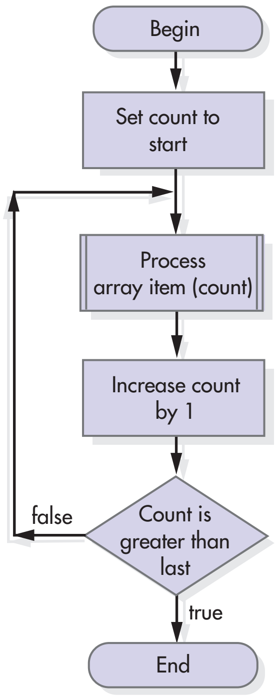


Figure 3.14 An algorithm that processes each of the elements of an array can be modified for any number of purposes.

For algorithms that have to be written from the start, programmers will often first work through the processing manually in order to identify the individual processes involved. These can then be translated into an algorithm. It is important that, for this stage of the development process, the programmer has a clear understanding of the data items that have to be manipulated. If the data items are not clearly defined in the programmer's mind, the processes chosen may not perform the required task or they may become too complex.

The user's perspective

Software applications are often written for a number of different users. The users perform their own distinct tasks, having individual requirements within the one application. The programmer has to provide the means for these needs to be met.

One of the user's basic needs is for the program to provide the processing necessary to complete the job. One of the most effective ways of achieving this goal is to have the user navigate through a number of levels to reach the required module of the program. This approach is a hierarchical one. In this arrangement, choices are given at a number of levels within the program. In using this approach, programmers force the user to exit through the same menus in the opposite direction in order to ensure that he or she does not become lost. In this way, the user's paths through the program are structured and predictable. Predictability brings comfort to the user.

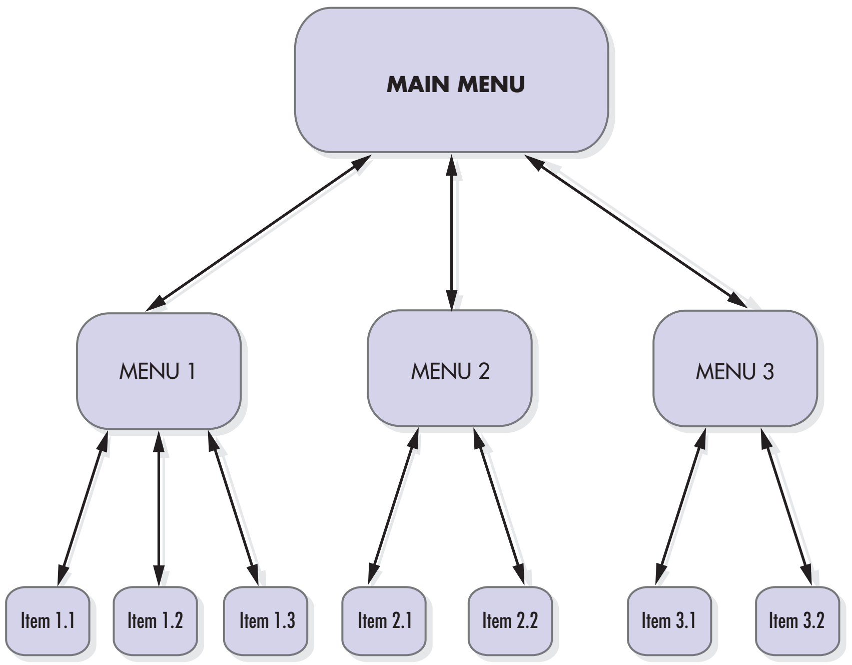


Figure 3.15 A hierarchical menu system allows users to know exactly where in the program they are.

Particular processes may require the user to interact using special peripheral devices. The manner in which these devices operate can also have a bearing on the way that the user perceives the software. For example, devices such as EFTPOS terminals have a very limited screen, so messages and prompts must be brief and to the point. Many users of these devices are also not computer literate, so the design of the interface needs to be carried out with this in mind.

Exercise 3.2

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

The design specifications cover the __________ of the project, __________ design, overall __________ design, __________ design, __________ design, a cross-reference with the __________ together with __________ and __________ of the final project. This specification lays down the __________ for the development as well as providing a yardstick that can be used to __________ the success of the __________ .

2 Describe the purpose of the design specifications.

3 Explain with examples why it is important for the developer to carefully choose the data types to be used in the program. What happens if the wrong data types are chosen?

4 Why do programmers choose a special name for each of the identifiers in a program? Use examples to illustrate your answer.

5 Describe the process of top-down decomposition. Explain why this process is widely used as a software development approach.

6 Explain how a software developer can cater for the needs of the user. Use an example to help illustrate your answer.

7 In Exercise 3.1 you examined a school report system. For this system, identify the data items and data types that will be needed.

8 Choose appropriate variable names for each of the data items you identified in question 7.

9 Describe the main processes that have to be carried out in the operation of the school report system. Some of these processes will be manual ones.

10 Describe the user requirements for the school report program. What special peripherals, if any, can be used as part of this system?

Modelling

A programmer must fully understand a problem before attempting to create a solution. Many tools can help with this process. Some of these tools are used to gain an overall picture of the system; others are used to help understand the processing; others are used to explain the interaction between parts of the system. No one tool can perform all these tasks. This section examines some of these tools and their uses.

Representing a system using diagrams

The old adage ‘a picture is worth a thousand words’ is certainly true in software development. A diagram on a single sheet of paper can fully describe a system, whereas a written description of the same system may take a number of pages. For example, the diagram in Figure 3.16 represents the way in which an address book program with password protection works. It is an example of a data flow diagram. Data flow diagrams show the relationship between the various modules and the data items that pass between them.

Input, process, output (IPO) diagrams

An IPO (input, processing, output) chart is used to describe the data elements that will enter the system (or subsystem), the processes that will occur and the data elements that will leave the system. Although the formats of IPO charts may differ, they will contain a heading, a list of inputs, a description of the processes and a list of outputs.

IPO diagrams provide a quick and easy method of investigating a system. Since they indicate the inputs, outputs and processing, they can be used to understand how data is being manipulated by the system.

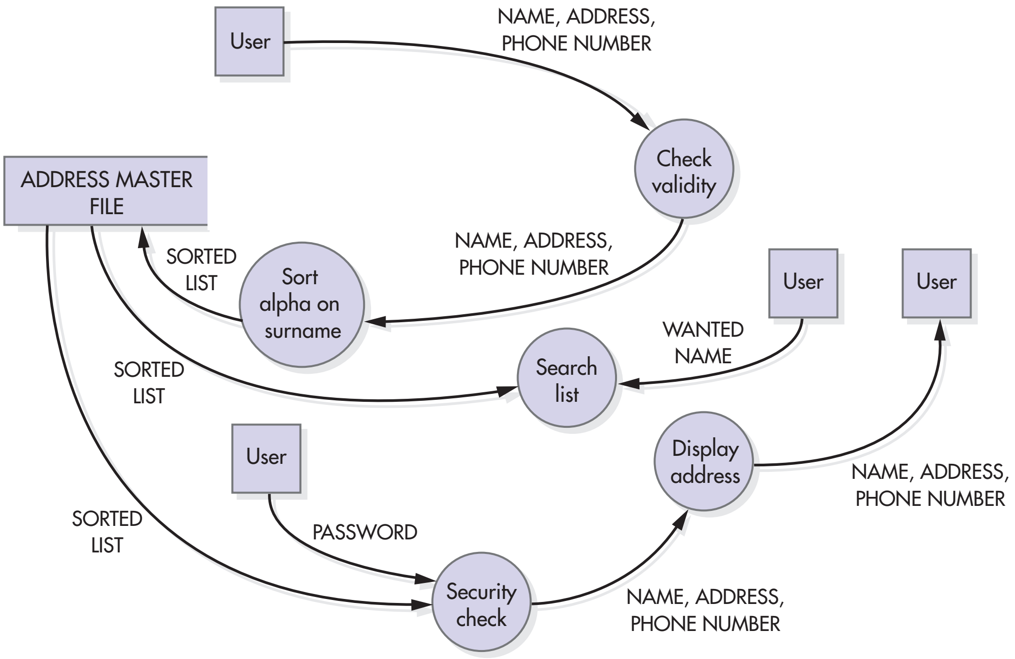


Figure 3.16 This diagram illustrates the interaction between the modules of an address book program.

IPO charts can also be drawn for any subsystem to show more detail, such as how each component manipulates the data. For example, Figure 3.17 shows the IPO chart for an automated greenhouse using a computer system to control temperature, light and water. This system can be broken further into three separate parts: there is one subsystem for temperature control (shown in Figure 3.18), one for the watering system (shown in Figure 3.19) and one for the light control.

IPO charts can help us determine how data is manipulated by the computer system, and when developed for subsystems of the main system can show how each component of a system manipulates the data.

A different IPO chart format is shown in Figure 3.20. This chart describes the processing of an employee's wages to determine the amount of tax to be deducted, the outputting of the net wage (after-tax wage), and the amount of tax payable. The form that the processing takes is shown in more detail than in the previous type of chart.

Input	Process	Output
Water Sunlight Electricity Plants Sensor readings Soil Fertiliser	Heat Light Moisture Growth	Growing plants Food

Figure 3.17 An IPO chart describing an automated greenhouse.

Input	Process	Output
Temperature Electricity	Reading sensors	Warmth

Figure 3.18 An IPO chart describing the temperature-control subsystem.

Input	Process	Output
Moisture reading Electricity Water	Reading sensors	Water to plants

Figure 3.19 An IPO chart representing the watering subsystem.

IPO chart
System: WAGES
Function: TAX, NET PAY CALCULATION

Input:	*Output:*
GROSS_WAGES	NET_WAGES TAX_AMOUNT

Process:

1. TAX_SCALE found from TAX_FILE
2. TAX_AMOUNT = GROSS_WAGES x TAX_SCALE
3. NET_WAGES = GROSS_WAGES – TAX_AMOUNT

Figure 3.20 An alternative form of IPO chart, describing a module that calculates the net wage and tax payable.

Storyboards

Storyboards are generally used for giving an overview of a program. They are particularly useful in multimedia productions and interactive programs where there are a large number of screens with complex patterns of navigation. They may, however, be used in simpler navigation systems as well.

Storyboards are useful for giving an idea of screen layout and of how groups or clusters of screens relate to each other. They also indicate the options available on each screen for navigation. Storyboard arrangements can be categorised as linear, hierarchical, network or hybrid.

Linear

In a linear arrangement, control passes from one subsystem to the next in a single sequence (see Figure 3.21).

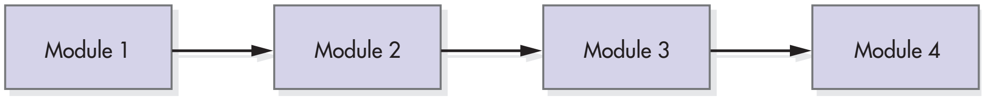


Figure 3.21 A linear storyboard arrangement.

A system such as a railway ticketing system may be represented by a linear storyboard arrangement if the system is viewed as the processes of ticket sales, ticket checking on admittance to the platform and collection of the ticket at the destination. It can be represented by the arrangement in Figure 3.22.

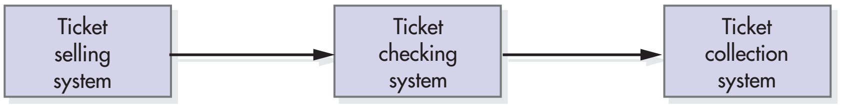


Figure 3.22 Representation of a railway ticketing system as a linear storyboard.

Hierarchical

In a hierarchical arrangement the subsystems are arranged as a tree. To reach a particular subsystem requires moving through each of the subsystems which are at a higher level in the tree (see Figure 3.23).

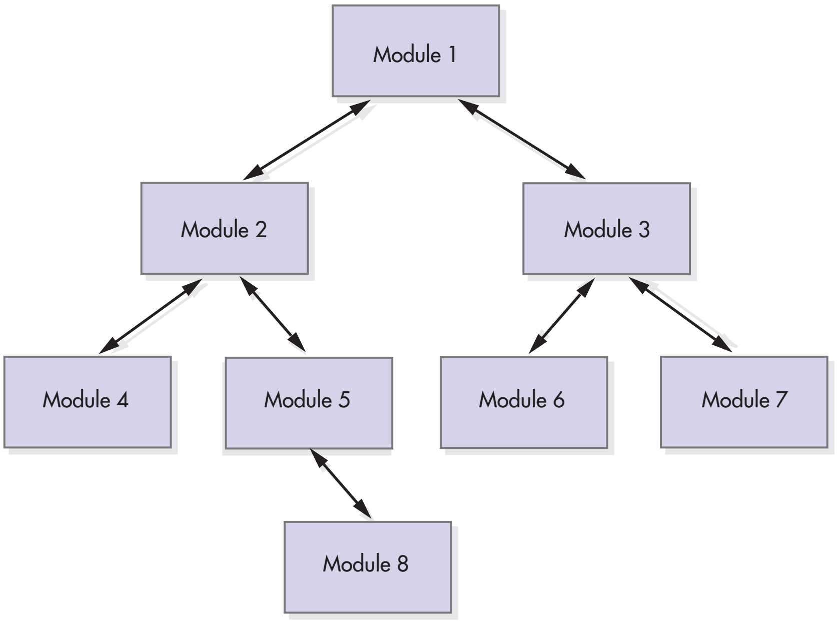


Figure 3.23 A hierarchical arrangement of storyboards.

A hierarchical storyboard arrangement could apply to a school administration system (Figure 3.24). The top level of the system is the main menu, each of the system's modules being accessed from this menu or menus at a lower level.

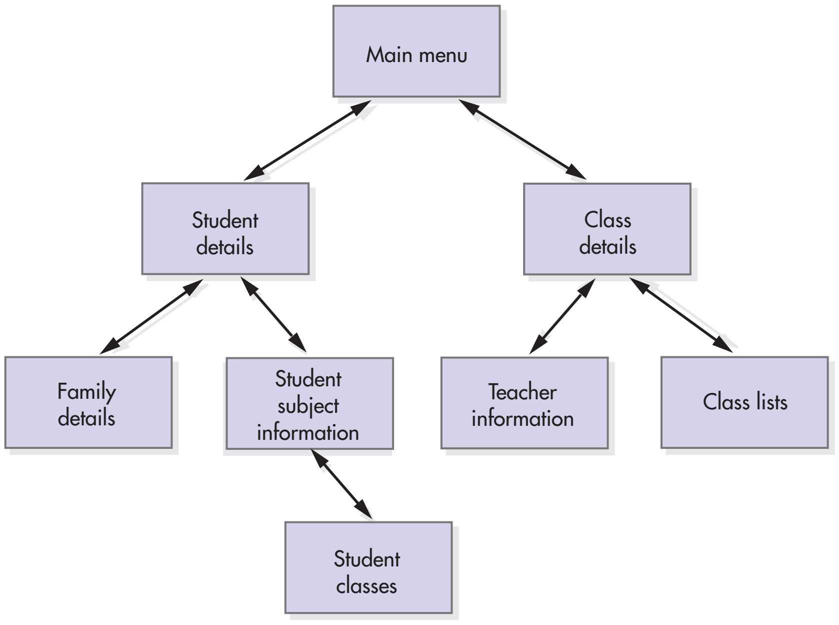


Figure 3.24 A representation of a school administration system arranged as a hierarchy of storyboards.

Network

In the network arrangement the subsystems are arranged as a web, in which modules may be directly accessed from other modules (see Figure 3.25).

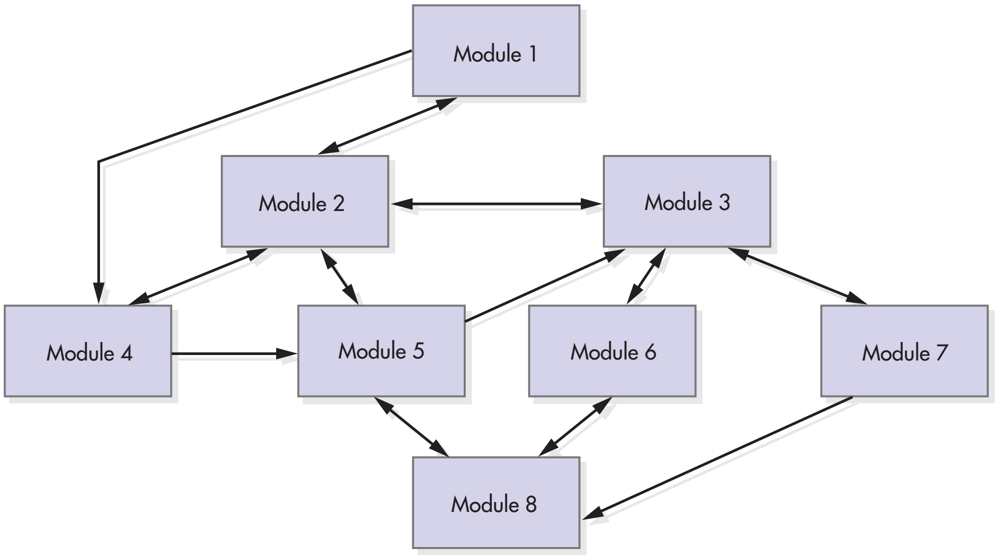


Figure 3.25 A network arrangement of storyboards.

An online library catalogue shown in Figure 3.26 is an example of a system that can be represented as a network. The user may wish to search for an item using a keyword (or keywords), an author's name or a subject. The search module will be basically the same, and then the book details and availability modules will become available. The user can trace back through the search to either refine the search criteria or review progress. This system leads to a small interconnected system.

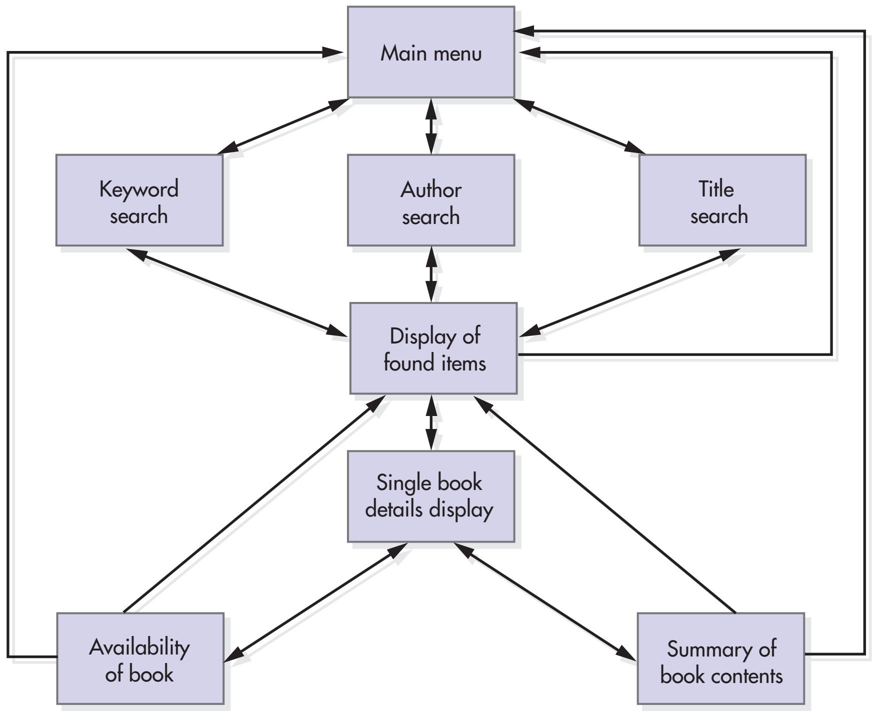


Figure 3.26 A representation of a library catalogue system using a network arrangement of storyboards.

Hybrid

A hybrid arrangement is a combination of two or more of the above types (Figure 3.27). A teaching and learning system can be represented by a hybrid arrangement of storyboards (see Figure 3.28). A linear system moves from topic to topic, and an interconnected system is used to assist with the correction of problems in any topic. If a student performs well in the pre-test, it is possible to pass on to the next topic without instruction. The lessons are placed in a sequence that is followed by the student. When a student has completed the post-test (following instruction), the results of the test are used to send the student either back to the lesson (if poorly understood) or on to the next topic in the sequence.

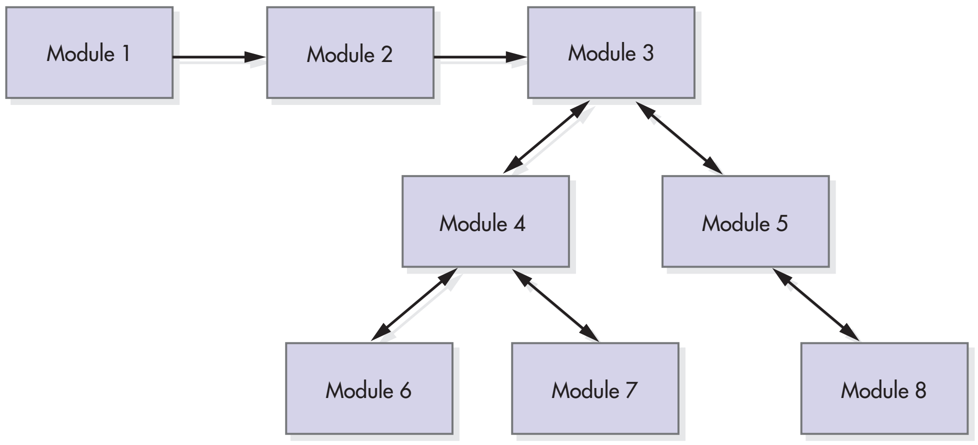


Figure 3.27 A hybrid arrangement of storyboards.

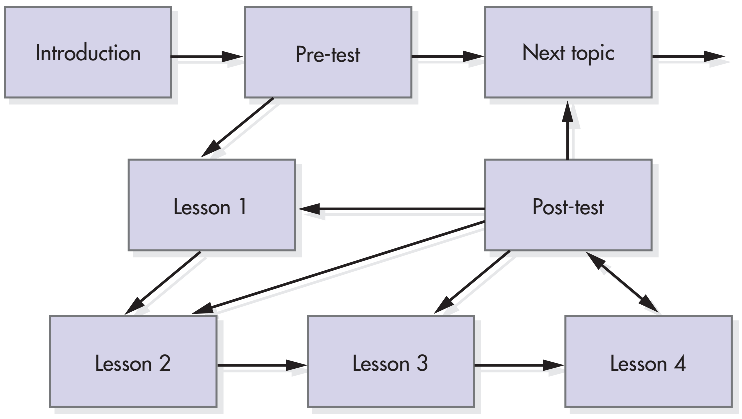


Figure 3.28 A representation of part of a teaching system.

Data flow diagrams

IPO charts show only the inputs, processes and outputs of a system, without showing relationships between them. Storyboards show the relationship between the screens within an application. Data flow diagrams show the flow or path of data throughout the system. They indicate how and where data is entered, stored, processed and output. Data flow diagrams are very useful for indicating where tasks overlap or where unnecessary storage or transmission of data occurs. Figure 3.29 shows the main symbols in data flow diagrams.

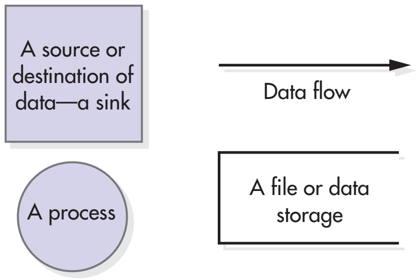


Figure 3.29 The symbols used in data flow diagrams.

A box is used to represent a source (the input of data into the system) or a sink (the output of data from the system). A circle represents the processing of data. An open rectangle represents storage of data within the system. Arrows are used to indicate the flow of data between parts of the system. Figure 3.30 shows the data flow diagram for a supermarket barcode scanner and checkout system. Note that data must always flow either to or from a process.

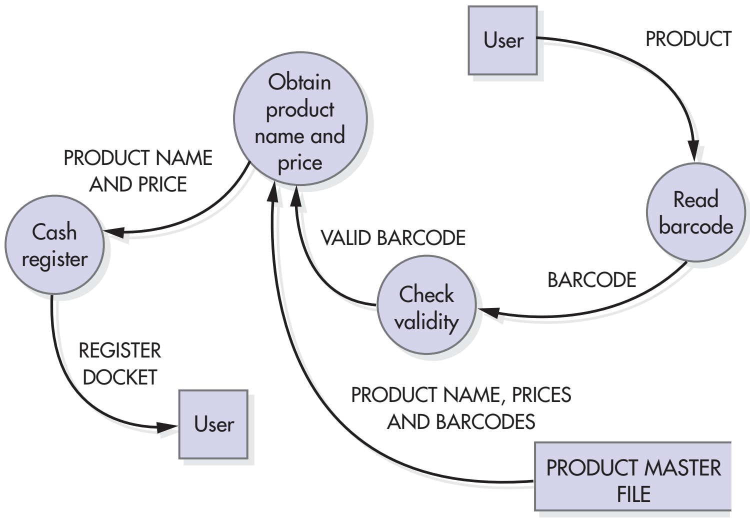


Figure 3.30 A data flow diagram for a supermarket barcode reader.

System flowcharts

The previous tools provide an understanding of only the data flow and the processes involved. To further understand an existing computer system and be able to identify possible improvements, it is necessary to have an understanding of the hardware being used and how it all interrelates. System flowcharts enable a graphical model of the physical system to be developed, indicating hardware devices, the storage medium and processing units.

Software developers need to understand the overall workings of the system that will use the software, in order to know where the software fits into the whole system. The standard symbols used in a system flowchart are shown in Figure 3.31. The system flowchart in Figure 3.32 shows the relationship between the components of an automated greenhouse.

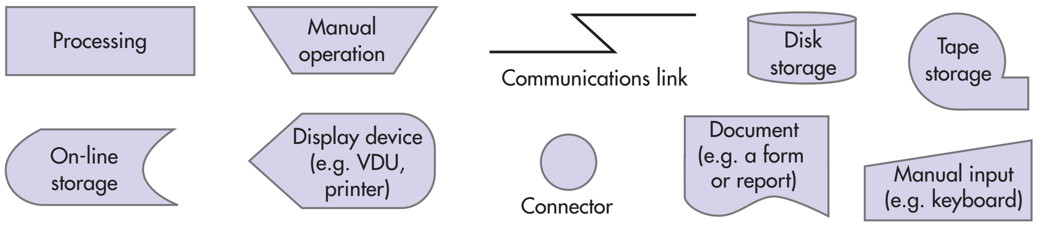

Processing
Manual operation
Communications link
Disk storage
Tape storage
On-line storage
Display device (e.g. VDU, printer)
Connector
Document (e.g. a form or report)
Manual input (e.g. keyboard)

Figure 3.31 The standard symbols used in a system flowchart.

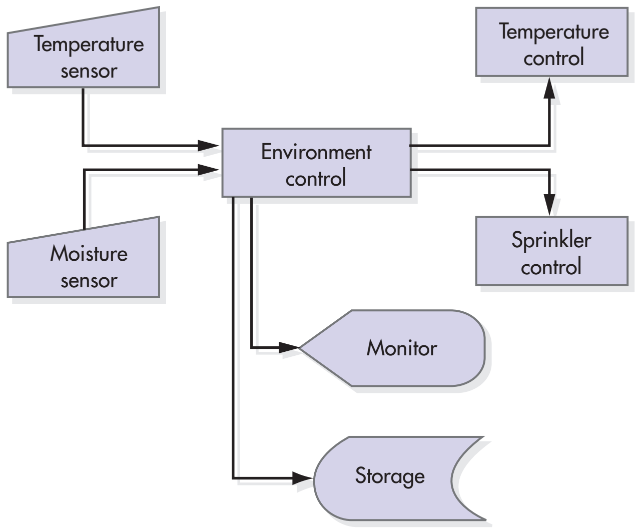


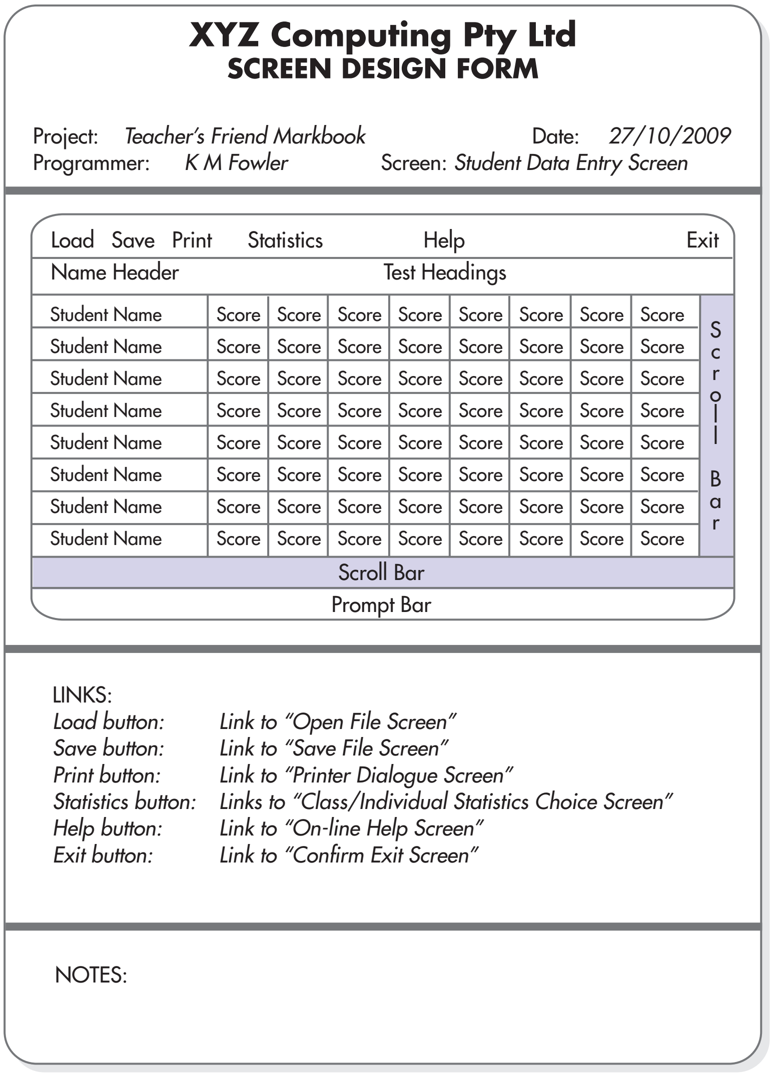
Figure 3.32 The system flowchart for an automated greenhouse.

Screen designs

The interface between the computer and the user cannot be treated as an afterthought. Before program construction begins, the screens need to be designed. This allows the programmer to integrate the screen design into the program right from the start, rather than having to integrate the design into an already-written module.

Screen design sheets will vary in layout. All sheets will contain an area for the actual screen design. Other areas of the design template will specify the links between that screen and other screens. The screen sheet will also contain a heading area that provides the details of the program, programmers, date of design and any other important aspects of the screen. CASE tools may also be used very effectively in the screen design process; integrated tools allow the design to be automatically introduced into the final application.

Screen layouts will also be included in the sections of the program documentation known as the input and output catalogues. These catalogue entries describe the purpose of each of the inputs and outputs as well as the form that it takes. These catalogues also include any report designs that are required by the program.


XYZ Computing Pty Ltd
SCREEN DESIGN FORM

Project: *Teacher's Friend Markbook* Date: *27/10/2009*
Programmer: *K M Fowler* Screen: *Student Data Entry Screen*

LINKS:

Load button:	*Link to "Open File Screen"*
Save button:	*Link to "Save File Screen"*
Print button:	*Link to "Printer Dialogue Screen"*
Statistics button:	*Links to "Class/Individual Statistics Choice Screen"*
Help button:	*Link to "On-line Help Screen"*
Exit button:	*Link to "Confirm Exit Screen"*

NOTES:

Figure 3.33 A screen design sheet will include the screen layout, together with other details important to its use.

Consideration of the use of a limited prototype

A developer can use a prototype as an effective development tool. As seen, a prototype can be developed into a full working solution. A prototype can also be used to gather further information about the way in which the system works.

When a prototype is used as a development tool it is designed rapidly using one or more CASE tools, often without regard to data validation or verification. The aim of the prototype is to determine how the system works, especially in the area of the human interface. The prototype fulfils this aim well. Since the user is presented with a working solution, valuable information is gained by the developer. This is especially true for interactive applications in which the contact between the user and the program is important. Prototypes of this kind are not suitable for applications that involve complex mathematical calculations, in part because input data is not verified or validated.

Other systems representations

A number of other methods of systems description methods are available to help with the design of a software application. Decision tables and decision trees help the programmer to understand the decisions that have to be made by a program. Structure diagrams are used to show the relationships between the various modules of a program.

Figure 3.34 Interactive games, such as *Half life*, may be developed with the aid of a limited prototype.

Decision tables and decision trees

The processes that operate on data in most information systems are determined by decisions. These decisions can take place in the computer, such as selection criteria or the scoring in a computer game, or they can take place outside the computer system, such as an operator accepting or rejecting applications. These decisions are made based on the data entered into the system according to a set of rules or conditions. Two tools used to display and analyse decisions are decision tables and decision trees.

A decision table indicates the alternatives for different circumstances based on the rules provided in the form of a table. They can be used to provide an understanding of the factors that affect data flow or processes in the system. For example, Figure 3.35 shows the decision table for the temperature-control subsystem for an automated greenhouse.

Temperature-control subsystem		**Rules**	
	1	2	3
Temperature is < 20	Y	Y	N
Temperature is < 10	N	Y	–
Turn one heater on	Y	N	N
Turn two heaters on	N	Y	N

Figure 3.35 Decision table for a temperature-control subsystem.

Decision trees indicate the decisions made within a system as the branches of a tree. They can be used in a similar way to a decision table, but they are easier to follow where decisions are based on results of previous decisions. Figure 3.36 shows the decision tree for the greenhouse temperature-control subsystem.

Structure diagram

A structure diagram is used to show the precise relationships between the various modules of a system. The chart employs rectangles to represent the modules, lines to represent connections and labelled arrows to represent the data items being passed between the modules. The use of a structure diagram allows program modules to be developed independently, as the exact forms of the input and output from these modules is known.

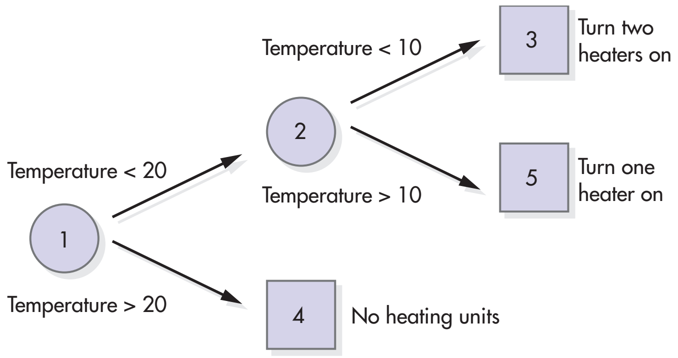


Figure 3.36 The decision tree for the temperature-control subsystem.

Structure diagrams are drawn to conform to a set of rules or conventions. Modules are represented by rectangles and are arranged from top to bottom with the controlling modules at the top. The order of execution of modules is from left to right. The symbols used and their meanings appear in Figure 3.37.

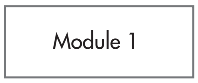
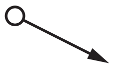
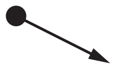
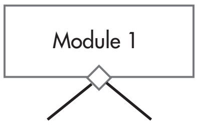
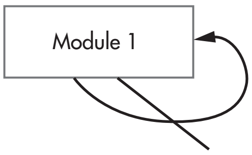
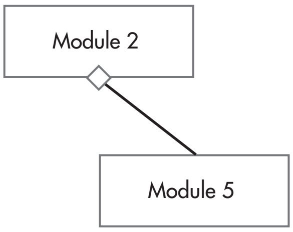
Symbol	Meaning
	A rectangle represents a module.
	The arrow shows the passing of a parameter.
	A filled arrow indicates a control parameter.
	A diamond indicates that a decision has to be made as to which module is executed.
	A circular arrow indicates repetition.
	An optional module is indicated by a diamond at the calling module. In this case, module 5 is optionally called by module 2.

Figure 3.37 Structure diagram symbols.

In the example in Figure 3.38, data item A is passed from module 1 to module 3, data item G is passed back to module 1, and data item B passes from module 3 to module 7. Module 8 passes data item H to module 5 which, in turn, passes the item on to module 2. Module 2 illustrates the use of a decision; it will either call module 5 and pass data item D to it or call module 6 in which case it passes data item E as a parameter. Module 1 repetitively calls upon module 4, passing data item C to it.

The structure diagram in Figure 3.39 represents a supermarket scanning system at a checkout. The scanned barcode is compared with the barcodes database in the supervising system, and the item's price is then sent to the till. At the same time, the item is deducted from the stock-in-hand database in the stock system. This process is repeated until all items have been scanned. After scanning, a total is calculated and printed on the customer's docket. When the total has been paid at the checkout, the amount is added to the day's takings transaction file for that checkout and the store.

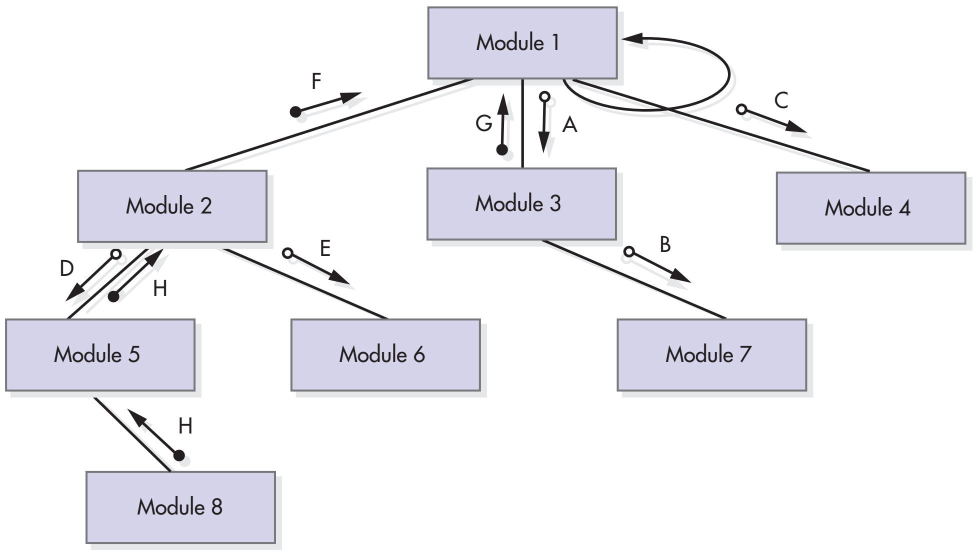


Figure 3.38 A structure diagram shows the relationship between modules, together with the flow of data between them.

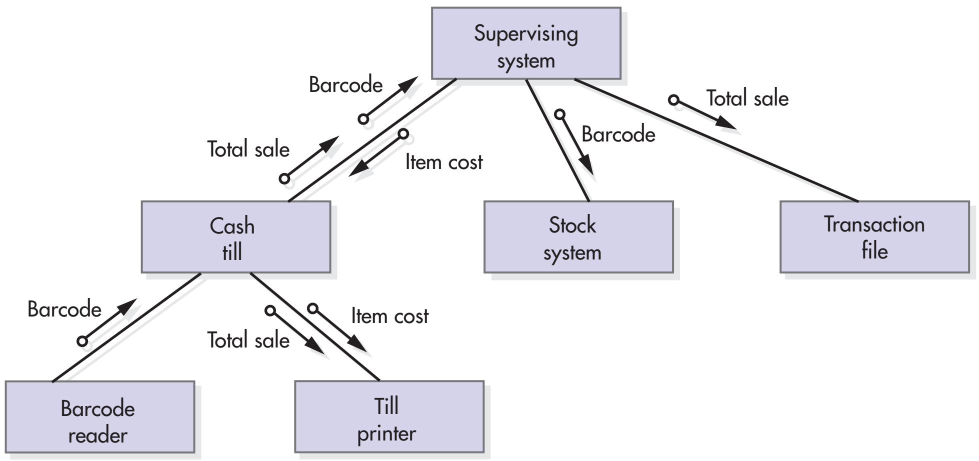


Figure 3.39 A structure diagram representing a supermarket checkout system.

Exercise 3.3

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

A number of tools can help the ____________ understand a problem before attempting to create a ____________ . Some of these tools give an overall ____________ of the system; others are used to help understand the ____________ ; others help explain the ____________ between ____________ of the system. Many of these tools use ____________ to explain the workings of the ____________ . Some of these tools are ____________ charts, ____________ diagrams, ____________ boards, ____________ flowcharts and ____________ designs.

2 Describe how IPO charts can be used to help understand the working of a system. Use an example to illustrate your answer.

3 Storyboards can be used to give an overview of the system. Describe an application where a storyboard would be useful. Give reasons why you would use a storyboard in this application.

4 Describe the different arrangements of storyboards in your own words. Give an example of an application of each type.

5 Explain, with an example, how data flow diagrams are used to understand the workings of a system.

6 Read the diagram in Figure 3.30 and describe in words how the supermarket barcode reader works.

7 Create a screen design sheet for the school report system discussed in earlier exercises.

8 Explain how a limited prototype can be used to gather information about a system.

9 Information is needed about the teacher-input module of the school report system discussed in earlier exercises. Use a simple database system, such as one found in an integrated application package, to create a limited prototype for the problem.

10 Create an IPO chart that describes the working of the teacher-input module of the school report system.

Communication issues

Implementation of a new system is always a great deal easier if those involved with the system feel that they have contributed to the changes. Many conflicts can be avoided if genuine communication is carried out between the developers and the users.

Developers are familiar with the technical aspects of a new system; the users are familiar with the operation of the current system. If these two perspectives can be joined in a harmonious way, the developers will benefit by creating a successful working system that is also accepted by the end users. Users will also benefit from this approach, since their experiences and concerns can be used to steer the development process to a successful conclusion.

The need to empower the user

Changing work practices are often cause for discomfort, resentment and fear. People who are going to be affected by these types of change will more readily accept them if they feel that they have had input into the process. People feel comfortable when they have some control of the situation and have a sense of ownership of the product.

Users will claim ownership of a software application that has been developed with their input. Ownership is especially important after installation, as users are more likely to describe problems or suggest enhancements if they feel that the software belongs to them.

Empowering the user also means giving them the ability to make decisions that affect their work. People need the mental stimulation that decision making provides. For example, workers in a fully automated factory are very easily bored by the task of supervising the machinery. They usually need to step in only when a machine has not performed satisfactorily and may have very little pride in what they do. On the other hand, people working in factories where the tasks are varied and decisions have to be made generally enjoy their work and take pride in what they do.

Software developers have a responsibility to ensure that the users of the software have input into its design and development and are given tasks that allow them to take full control of their activity.

The need to acknowledge the user's perspective

The software developer has the technical knowledge about how to translate the wishes of the user into properly functioning code. The user is the source of knowledge about the functioning of the system. The developer and the user are equally important to the success of the software project, and the developer must accept the expertise of the user and use it to better understand the system.

Also, the users will be managing the software and its interface long after implementation, so their perspective should be taken into consideration when designing both the processes and the interfaces.

Enabling and accepting feedback

For an effective system of communication, channels must kept be open. Developers need to establish both formal and informal channels of communication. Formal channels will include documents such as memos and regular meetings to keep the users up-to-date with the development process. Informal discussions should also be possible at any time during the development process, as quite often matters will arise that cannot be kept until the next formal meeting. The users should feel free to contact the developer with any concerns and the developer should have easy access to the users.

The development process will proceed more smoothly if people can receive constructive criticism without being offended. The software developer and the customer need to be able to trust each other not to take constructive criticism personally. In this way, an effective communication channel can be established and maintained throughout the whole of the development process.

Exercise 3.4

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Developers need to establish both ________ and ________ channels of communication. Formal channels will include regular ________ to keep the users up-to-date with the ________ process and ________ such as memos. If concerns arise between formal meetings, then ________ communication can take place.

2 What is meant by the term 'empowering the user'? Use one or more examples to illustrate your answer.

3 Why is the user's perspective important to the software developer? How can the developer make sure that the user's perspective is taken into consideration?

4 Explain why feedback is important to the software development process. Use examples to illustrate your answer.

5 What steps could be taken to address the communication issues during development of the school report system discussed in earlier exercises?

Review exercises

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

In order to solve a problem, we need to ________ it. A study, known as a ________ study, will examine a number of ________ solutions and determine which can be ________. Software developers also use ________ such as data flow ________ to help understand the problem. Communication between the ________ and ________ is also important to the ________ of the project. Both ________ and ________ communication needs to take place.

2 Choose the alternative, A, B, C or D, that best answers the question.

a A DNA recognition system is proposed for students to sign in at schools. The school investigated the possible systems and found that the equipment needed was going to take 5 minutes to process each student. They decided not to go ahead with the system. This is an example of
   A financial feasibility
   B operational feasibility
   C schedule feasibility
   D technical feasibility

b The data type most suitable for storing the daily takings of each of the six cash registers in a store over a period of a month is
   A an array of characters
   B an array of integers
   C an array of real numbers
   D an array of strings

c A data flow diagram is used to show the
   A relationship between physical elements of the system
   B inputs, processes and outputs of the system
   C processes and movement of data
   D an overview of the program design

d A limited prototype is used to
   A solve the problem
   B gain information
   C examine how the user interacts with the program
   D test user interfaces

e The end-user should be consulted on program design because
   A the user knows the operation of the current system
   B the user may resent the new system
   C to provide the user with a sense of ownership
   D all of the above

3 Describe the constraints that may affect the development of the school reporting system described in earlier exercises of this chapter.

4 Explain how you could involve each of these users in the development of a school report system: teachers, school clerical staff, school executive, students and parents.

5 Construct a data flow diagram that shows the workings of a school report system.

6 Design a storyboard arrangement that describes a system for gaining a driver's licence.

7 Describe the operation of your school library's catalogue and borrowing system by means of one or more of the systems description methods discussed in this chapter.

8 Examine a system of your own choice and show its operation by means of appropriate systems description methods.

Team Activity

Your school wishes to implement the school report system described in exercises earlier in this chapter. Your team has to prepare a feasibility study for this project. You will have to examine all aspects of feasibility in the report. You will also need to suggest the most feasible solution and justify your choice. The report should be presented in a suitable form.

Chapter summary

- The three stages of software development are understanding the problem, working out the solution and checking the solution.
- Defining the problem involves examining the needs of the users, the objectives of the solution and the boundaries within which the solution has to operate.
- Needs include those of the user, data, storage, processes and output.
- The feasibility of the solution has to be investigated.
- Feasibility involves whether the problem is worth solving, constraints on the project, financial feasibility, operational feasibility, technical feasibility, scheduling feasibility, feasibility of alternatives and the social and ethical considerations.
- The design specification sets the guidelines for development and provides a yardstick for evaluation.
- The design specification covers the scope of the project, data design, overall program design, interface design, cross-referencing with the requirements definition, and testing of the final product.
- The developer's perspective of the program is in terms of data and processes.
- The programmer has to choose the data type and design the variables and algorithms.
- Users look at the program as a solution to their problem. This means that it has to provide the processing necessary to perform the job.
- Systems can be represented using diagrams such as IPO charts, storyboards, data flow diagrams, system flowcharts, screen designs, prototypes, structure diagrams, decision tables and decision trees.
- Empowering users provides them with a sense of control over their situation.
- Involving the user in the software development process gives the user a sense of ownership of the software.
- The developer should acknowledge the user's perspective by looking for, and accepting, input into the development process.
- Developers and users can both benefit from open communication during the development process.

chapter 4

Planning and design of software solutions

Outcomes

A student:

- identifies needs to which software solutions are appropriate (H 4.1)
- applies appropriate development methods to solve software problems (H 4.2)
- applies a modular approach to implement well-structured software solutions and evaluates their effectiveness (H 4.3)

Students learn about:

Standard algorithms for searching and sorting

- finding maximum and minimum values in arrays
- processing strings (extracting, inserting, deleting)
- linear search
- binary search
- bubble sort
- selection sort

Custom-designed logic used in software solutions

- identification of inputs, processes and outputs
- representation as an algorithm
- definition of required data structures
- use of data structures, arrays of records
- thorough testing

Standard modules (library routines) used in software solutions

- identification of appropriate modules
- consideration of local and global variables
- appropriate use of parameters (arguments)
- appropriate testing using drivers

Students learn to:

- recognise the logic in a standard approach (such as a sort or search)
- apply standard approaches as part of the solution to complex problems
- document the logic required to solve problems
- develop a suitable set of test data and desk check algorithms that include complex logic
- select an appropriate data structure to solve a given problem
- develop a standard module and document its use
- correctly incorporate a standard module into a more complex solution, passing parameters effectively

Developing a problem-solving strategy

How best to learn programming

Many subjects are 'read and recite' in nature. You read and study the material, summarise it, internalise it and eventually regurgitate it in an examination or an assessment task. Programming is different. Programming, and algorithm design, can only be learnt by lots of practice. It is only understood when you actually devise a solution to a problem using your own skills and experience. Also, a correct solution to an algorithm design problem is rarely created in one attempt. You have to create a trial algorithm, test it with some sample data and compare the results against some known or expected outcomes. The algorithm usually has to be modified and tested again, often many times, before the solution is really correct. With this in mind, you are encouraged to attempt to solve as many problems as possible. You need to internalise the processes of trialling, testing and modifying your algorithms until they are working properly.

This chapter contains examples and exercises which will help you to practise and gain this experience. The chapter also emphasises design patterns which you should understand. It is a good idea to keep a separate notebook into which you can record these design patterns. Each recording will consist of a fragment of an algorithm that describes the design pattern, a brief summary of how it works and the kind of problems in which it is useful. The recording could also include an example algorithm that employs the design pattern. As your experience grows, and the notebook contains more patterns, you will begin to formalise your skills. Of course, the ultimate situation is when the patterns become completely internalised in your mind. You can start your notebook now by recording the 'sentinel-controlled, pre-test loop' design pattern shown later in this section.

Design patterns

Expert programmers achieve their success by using design patterns. A design pattern is a way of combining programming structures into an easily identified and highly reusable group. These patterns, once learned, can be used again and again in the solutions to problems.

Experienced programmers find that any problem they are given is made up of many familiar design patterns. That is, every problem will contain parts that are similar to tasks the programmer has done before, so applying known design patterns solves most of the problem. Any new concepts or unfamiliar aspects of the problem need to be analysed and new structures created to solve them.

An example of a simple design pattern is presented here. Consider the task of processing information as it arrives from an input stream. Often such input streams contain a special value that signifies the end of the data. These situations are common and a design pattern called a 'sentinel-controlled, pre-test loop' can be applied to them. For example, when you are paying for groceries at a supermarket you usually place a length of wood across the conveyor belt to separate your goods from the following customer. The block of wood is a sentinel. It is seen by the checkout operator as a signal that the processing of your shopping can stop. Below is a pseudocode version of an algorithm that could be used by a checkout operator in this situation.

```
BEGIN process shopping items
    get the first grocery item // priming read
    WHILE ( the item is not the separator )
            process the next item
            get the next grocery item // follow-up read
    ENDWHILE
END process shopping items
```

Note here that the algorithm fetches a new item immediately before testing the condition on the loop, first with the priming read before the loop starts and subsequently as the last instruction in the body of the loop just before control returns to the loop test.

The above algorithm just processes shopping items. What is really important is the design pattern being used. In general and reusable form, the design pattern can be described as shown in the algorithm below:

```
// Design Pattern: Sentinel-controlled, pre-test loop
// Usage: to process a stream of data with a sentinel value

read the first datum
WHILE ( the datum is NOT the sentinel )
   process the datum
   read the next datum
ENDWHILE
```

This is worth recording so you can apply it whenever the need arises. At the end of this section some strategies for recording design patterns are presented. The one above could be the first entry in your personal design pattern collection. In the rest of this chapter you will be presented with many useful design patterns. If you learn them and get plenty of practice applying them they will become part of your programming repertoire. You will then start to become an experienced program designer.

Structured programming and functional decomposition

It is possible to write an algorithm from start to finish as one continuous flow of instructions. Writing an algorithm this way has several disadvantages:

- The algorithm is usually long and takes a long time to read, understand and debug.
- There is no easy way to get an overall view of what the algorithm does.
- There is little opportunity to take sections out of the algorithm and use them in other solutions.
- Instructions that occur several times at different places in the algorithm need to be re-written each time they occur.
- The whole solution usually becomes less robust as it is modified.

Algorithms written as one long sequence of instructions are sometimes called 'spaghetti code'. They often look like tangled cooked spaghetti. It is impossible to see any real purpose in them and they have no identifiable structure. What is worse, when they are implemented in a programming language and put into service, the entire program tends to become brittle, fragile and prone to failure as it is continually modified.

In contrast, structured programming is the technique of writing programs or algorithms that have a recognisable structure. The structure is provided by having a general or overall solution to the problem presented first, called a main module, followed by a collection of sub-modules which are activated by the main module to perform a well-defined part of the work. The main module removes the first two disadvantages listed above. It is short and easy to read, and it gives an overall summary of the solution.

At the heart of structured programming is the use of sub-modules or 'functions' that are activated by the main module. The technique of breaking a large or complex problem down into functions is called 'functional decomposition'. The functions can be designed and written separately, can be tested separately and should be independent of each other. Useful functions can encapsulate whole design patterns and thus be used again and again in the solutions to many different problems. This removes the third disadvantage from the list. A function written only once can be activated, or 'invoked' many times, thus saving writing time and unnecessary repetition of code, which removes the fourth disadvantage. During program maintenance, modifications can be made to some functions without having to modify the whole algorithm—helping to remove the fifth disadvantage. Functions are discussed in detail later in the chapter.

A worked example of structured programming

This example demonstrates the difference between a 'spaghetti' algorithm and a structured one. The problem is small and simple. Although the spaghetti version is readable, you should concentrate on the structural differences between the versions.

PROBLEM STATEMENT

A rail ticket pricing machine displays the cost of a rail ticket when a person enters two values—a destination and an age category. The machine can calculate the costs of many tickets. When a railway officer enters a destination of 'Close', the machine displays the total cost of all the tickets priced and then shuts down.

There are four destination values that can be entered—'City', 'Interstate', 'Suburban' and 'Close'. There are two age codes—'adult' and 'Child'.

The ticket costs are based on the values in Table 4.1.

	City	Interstate	Suburban
Adult	6.40	35.00	4.10
Child	4.00	12.00	1.00

Table 4.1

Write a pseudocode description of an algorithm that the machine could use.

SOLUTION 1 (NOT STRUCTURED)

This solution flows from start to finish in one module. The two loop keywords 'while' and 'endwhile' are a long way apart in this algorithm and so it is quite difficult to see at a glance where the loop starts and ends.

```
BEGIN rail ticket costing
    set variable totalCost to zero
    create variable cost
    get the destination value into variable dest
    WHILE( dest is NOT 'Close' )
        get age code into variable ageCode
        IF( dest = 'City' )
            IF( ageCode = 'Adult')
                set cost to 6.40
            ELSE
                set cost to 4.00
            ENDIF
        ELSE
            IF( dest = 'Interstate')
                IF( ageCode = 'Adult')
                    set cost to 35.00
                ELSE
                    set cost to 12.00
                ENDIF
            ELSE    // must be suburban destination
                IF( ageCode = 'Adult')
                    set cost to 4.10
                ELSE
                    set cost to 1.00
                ENDIF
            ENDIF
        ENDIF
        add cost to totalCost
        get the destination value into variable dest
    ENDWHILE
    print totalCost
END OF ALGORITHM
```

The next version employs functional decomposition to do the same work.

SOLUTION 1 (STRUCTURED)

Since the pricing machine processes an arbitrary number of tickets, it needs to employ a loop. Since there is a special destination that signals the end of the input data, the loop will be sentinel controlled. The design pattern introduced above is thus quite appropriate for this problem.

The overall work of the algorithm is to price many tickets, so we can make this our main module.

```
BEGIN main module: ticket pricing
    set variable totalCost to zero
    create variable cost
    get destination into variable dest
    WHILE( dest is NOT 'Close' )
        get ageCode
        set cost to calculateCost with dest and ageCode
        add cost to totalCost
        get destination into dest
    ENDWHILE
    print totalCost
END
```

The start and end of the loop are close together here. They are easily seen in the one glance. This makes the module easier to understand. The actual calculations of the cost of each ticket are not included in the main module but have been written as a separate sub-module or function. The underlined instruction

calculateCost with dest and ageCode

indicates where the sub-module is activated. Note that the values of dest and ageCode are sent to the sub-module so it can calculate the correct price. Here is the sub-module in pseudocode.

```
BEGIN sub-module: calculateCost with dest and ageCode
    IF( dest = 'City' )
        set cost to cityPrice with ageCode
    ELSE
        IF( dest = 'Interstate' )
            set cost to statePrice with ageCode
        ELSE
            set cost to suburbCost with ageCode
        ENDIF
    ENDIF
    return cost // value is sent back to the code which
                // called this module
END sub-module
```

This module contains two nested selections. It avoids the complication of dealing with the actual ageCode calculations by passing that task off to other modules. This has been exaggerated in this example to emphasise the functional decomposition technique, but it does have the benefit of allowing the module to concentrate on just one well-defined task.

The other sub-modules are given below without comment.

```
BEGIN sub-module: cityPrice with ageCode
    IF( ageCode = 'Adult' )
        return 6.40
    ELSE
        return 4.00
    ENDIF
END sub-module

BEGIN sub-module: statePrice with ageCode
    IF( ageCode = 'Adult' )
        return 35.00
    ELSE
        return 12.00
    ENDIF
END sub-module

BEGIN sub-module: suburbPrice with ageCode
    IF( ageCode = 'Adult' )
        return 4.10
    ELSE
        return 1.00
    ENDIF
END sub-module
```

DISCUSSION

The style and structure used in the second algorithm is what will be used in this chapter. You may be thinking that the first algorithm is preferable, as it does appear to be shorter and the second one does appear to be much more complex.

As problems get bigger and more complex, and as they require more sophisticated solution algorithms, so the need for the structured approach will increase. You are strongly advised to study the techniques of structured programming carefully and to practise the technique in your own algorithms.

To complete your study of this example, look at the structure chart below. It shows in graphical form how the modules relate to each other.

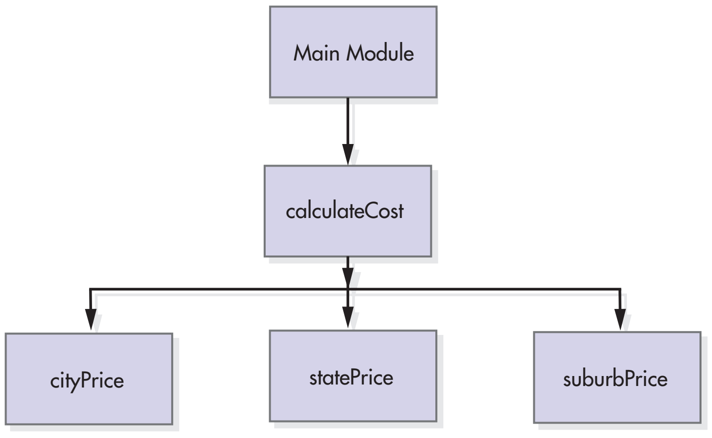


Figure 4.1

Exercise 4.1

1. What are some of the keys to learning to write algorithms and programs?
2. What is a 'design pattern'?
3. Successful programmers have a large and well-tested personal repertoire of design patterns. How does this collection of design patterns help to make a programmer successful?
4. What is functional decomposition and how does it relate to structured programming?
5. List some of the disadvantages of writing a program as one long sequence of instructions (that is, without any functional decomposition).
6. For small problems, a single sequential algorithm is often easier to write and understand than a structured solution that contains several modules. Why does this situation change when large problems are solved?

Design concepts

Identification of appropriate modules

The structured algorithm shown in the example on p. 78 is best understood by reading its main module first, which gives the overall view of the purpose of the algorithm. Once the main module is understood, the reader can then view the sub-modules to get further and more detailed information about what the algorithm does and how it does it.

Algorithms are also usually written by designing the main module first, then designing the sub-modules. Such a design process is called a 'top-down' method and is the most popular way to solve problems with structured programs.

It is equally valid, although less common, to work the other way around. That is, the minor parts of an algorithm are designed first, then when these are written and tested they can be combined to form the overall solution.

Think about how you solve a large jigsaw picture puzzle, one with many pieces. First you might find the corner pieces and all the other pieces that have straight edges. By fitting these together you can form the rectangular boundary of the whole puzzle. Then you can start to fit in all the other pieces by finding those that mesh with the boundary. The puzzle is solved from the outside inwards and, as each piece is added, the picture is formed.

It is equally likely that some complete sections of the jigsaw puzzle will be completed separately. All the light-blue pieces making up the sky might be joined in one section, or a large tree or a house might be developed on its own. This technique is regularly used when several people are working together on a really large jigsaw puzzle; each group works on a separate section. When some of the sections are complete, or almost complete, they can be fitted together to make more of the picture visible.

Designing and writing algorithms can be done in the same way as the jigsaw puzzle: by defining the main module first, followed by the smaller parts (the 'top-down' approach), or by completing several well-defined sub-modules separately first and then combining them (the 'bottom-up' approach). You will choose the approach that suits you best. You will probably use the top-down method for modest to large problems and may adopt the bottom-up approach when working as a member of a team or when faced with a huge and complex problem. Whichever approach you take, the final algorithm will still be structured and employ functional decomposition. Also, anyone who reads your algorithm will start by reading the main module then look at the sub-modules, even if it was not constructed that way.

Identification of inputs, processes and outputs

The discussion so far has dealt only with the processing or 'doing' side of algorithm design. We must also be concerned with the information our algorithms process—the data. We need to know how to manage this data, where it comes from, how and where to hold it, where it is allowed to be used and how to format it for output.

In the Preliminary Course you saw that for any problem there should be a clear statement of what the input data is and what the output data is. A given problem will also specify what processing needs to be done, although not necessarily 'how' to do it. You may have actually listed some input and output specifications during the analysis phase of your algorithm development, as shown in the following example.

Example

PROBLEM STATEMENT

Write an algorithm that accepts a gross salary and calculates the tax payable on it and the net salary.

INPUT, OUTPUT AND PROCESS

These can be listed in a table.

Entity	Description	Details
Input	a gross salary	a real number (dollars and cents)
Output	tax payable net salary	a real number (dollars and cents) a real number (dollars and cents)
Process	Calculate the tax payable. Calculate the net salary.	Multiply the gross salary by the tax rate to calculate the tax payable. Subtract the tax payable from the gross salary to calculate the net salary.

Table 4.2

Consideration of local and global variables

In an algorithm description data values are held in variables. Variables are just short names for the quantities they represent. Values can be stored in variables in several ways, such as:

```
// simple assignment within an algorithm
set grossSalary to 23900
// calculation and assignment
set netSalary to grossSalary × 0.48
// value is obtained interactively from a user
get grossSalary
// value is read from an input file
read grossSalary
```

The scope of variables

Although you are probably already familiar with the use of variables as shown above, it is also important to know where variables can be created and used. The preferred rule is that variables can only be used in the module in which they are created. This is certainly the case in most modern programming languages, and although you may not be coding actual programs it is best to adhere to that same policy in your algorithms.

The policy means that, if you create a variable in your main module, you cannot expect any of the sub-modules to be able to access its value; its use is restricted to that main module. The same applies to variables created in sub-modules. The official term used to describe where a variable can be used is 'scope'. We say that the 'scope' of a variable is the part of an algorithm where it can be used. The idea of restricting the scope of variables to their own modules may seem a little strange at first. You might be asking yourself how sub-modules can do useful work if they cannot access data elsewhere. This idea will be discussed in detail later in the chapter, 'Modules and functions'.

Example showing variable scope

The sample problem given above, asking for a tax and net salary calculation, can be used to show how data is passed from one module to another and how the scope of variables is restricted. Below is a main module for the algorithm.

```
BEGIN Salary calculations (main module)
   get the value of grossSalary   // (1)
   set taxValue to calculateTax with grossSalary  // (2)
   set netSalary to calculateNet with grossSalary and taxValue
   print 'Tax is ' taxValue
   print 'Net is ' netSalary
END Salary calculations
```

Notice in the main module above that we have created a variable called 'grossSalary' on line (1) and assigned some input value to it from a user. This variable belongs to the main module; none of the other modules know about it.

Obviously the `calculateTax` sub-module needs to have a gross salary to do its work, but how can the sub-module get this information if it is restricted to the main module? The answer is seen by looking carefully at how the sub-module is called on the line marked (2):

`calculateTax with grossSalary`

We use the special word 'with' to show that when the calculateTax module is called, the value in the grossSalary variable *is passed to it*. On the line marked (2) we have created a variable called 'taxValue'. This variable is assigned a value that the sub-module 'calculateTax' returns. In summary, we can say that the sub-module 'calculateTax' accepts a gross salary as its input, and returns a tax value as its output. Here is the algorithm for the calculateTax sub-module, it should make the explanations above clearer.

```
BEGIN with grossSalary
    set tempValue to grossSalary × 0.45  // (1)
    return tempValue  //(2)
END
```

On line (1) in this module we have created a local variable `tempValue`. This variable is useable only inside the module; no other module knows about it. It is called a 'local variable' and has 'local' scope. The answer to the tax calculation is stored in this local variable on line (1) and then, on line (2), the value is sent back to the main module.

The algorithm for the calculation of the net salary is similar. Its input is a gross salary and a tax value. It subtracts the tax from the gross salary and returns the answer as its output.

```
BEGIN calculateNet with grossSalary and taxValue
    set tempValue to grossSalary – taxValue
    return tempValue
END
```

Local variables are those created within modules. They are usually created to help with some calculation or to temporarily hold some information during the running of the module. The `calculateNet` module above is so simple that we could write it without needing the local variable `tempValue`, like this:

```
BEGIN calculateNet with grossSalary and taxValue
    return grossSalary – taxValue
END
```

The 'Modules and functions' section of this chapter contains further details about how modules are called, how data is passed to them and how data is returned from them.

Global variables

Some programming languages allow us to create variables that can be accessed by all modules at the same time without the need to pass their values around. For these variables the scope is the entire algorithm. These variables are called 'global variables'. Allowing all modules to access the same global variable can be quite efficient in programming, as less data needs to be passed around from module to module. However, it often results in errors. If a global variable has its value changed and becomes inaccurate, it is very difficult to tell which module is at fault since they all have access to it. The sample algorithms in this chapter do not use global variables.

Thorough testing

You can never be sure that an algorithm is correct until you test it, and proper testing cannot be done by just reading the algorithm. To test an algorithm properly you must provide some sample input data values, run your algorithm using this data and check that the output is correct. The only way of knowing that an output value is correct is to have calculated it manually beforehand.

Desk checking

Desk checking is done by following your algorithm line by line and doing what its instructions say. When desk checking, pretend you are a computer (a high-speed moron) and do exactly what the instructions say—no more and no less. Correct desk checking requires that you keep track of the values of all variables, writing down their names and values and adjusting the values as the algorithm progresses.

You need to do several, perhaps many, desk checks on most algorithms. Each desk check uses different sample data. The sample data must be such that all branches in the algorithm are exercised and the conditions of all loops are tested.

The following fragment contains a simple nested selection.

```
IF cost > 20
THEN
    print 'too expensive'
ELSE
    IF cost <= 5
    THEN
        print 'too cheap'
    ELSE
        print 'just right'
    ENDIF
ENDIF
```

To test this fragment thoroughly there should be at least five sample values chosen for the cost variable. One value should be greater than 20, one should be less than 5 and one should be between 5 and 20. The other two values should be exactly 5 and exactly 20, the boundary values for the selection.

These sample data and the results obtained for them can be set out in a short table, as in Table 4.3. Verify these results yourself by actually desk checking the above algorithm. The last two sample values are the boundary conditions and finding their output is left as an exercise for you.

Cost	Output
40	too expensive
15	just right
3	too cheap
5	???
20	???

Table 4.3

Trace tables

A good way to keep track of the values of your variables is to use a trace table. A trace table has at least two columns. The first column contains a list of the variable names, one variable name per row. In the second column the values of the variables are written next to their names. A third column can be used for optional comments. As you desk check your algorithm, you update the value of the variables in the trace table according to the instructions given in the algorithm. Remember to do this without any guesswork and only make changes that the algorithm dictates. A trace table can also include a row for final output, such as material printed or written to files by the algorithm.

A simple trace table can be constructed for the tax calculation example. There are four variables to trace plus final output, so the table has five rows.

Variable	Value	Comments
grossSalary		
taxValue		
netSalary		
tempValue		
Printed output		

Table 4.4

If we take $20 000 as a sample input data value, we can write this into the cell next to the `grossSalary` variable. As we trace the algorithm, the values of the other variables are filled in. Table 4.5 shows a completed table with a $20 000 gross salary.

Variable	Value	Comments	
grossSalary	20000	from user	
taxValue	9000	returned to main	
netSalary	11000	returned to main	
tempValue	~~9000~~	in sub-module calcTax	20000 × 0.45
	11000	in sub-module calcNet	20000 – 9000
Printed output	Tax is 9000		
	Net is 11000		

Table 4.5

The value of 9000 is crossed out in the `tempValue` row because this variable was created and used in both modules. When the `calculateNet` module used it, its old value was changed, as shown in the comments column. A neatly printed trace table like the one in

Table 4.5 is not really an accurate representation of a trace table. Trace tables are completed by hand with a lot of crossing out and editing as the values of the variables are amended during the desk checking of the algorithm.

Field testing

Before the actual computer software is delivered to its end users it too must be tested thoroughly. The testing includes desk checking by the programmers (as seen above), in-house testing by the development team, and field testing which involves giving the completed software to a representative group of users so they can try it out under real working conditions. Any problems or errors identified in the testing phase are then remedied by making modifications to the software before it is finally delivered.

This testing is expensive, both in time and money. Sometimes software is found to contain bugs after it is released. When end users or customers find such problems they report the problems to the vendor. After modifications are made to remove the errors, a new release of the software is made using a higher version number. This is field testing in the extreme. Releasing software too soon forces paying users to put up with bugs. It is frowned upon in the software development field but has become a fact of life as vendors rush to be first to the market with their latest software products.

Prototyping

A prototype is a partially complete software product which is presented to a small group of users so that feedback can be provided to the developers. A prototype does not contain all of the functionality intended for the finished product, but it often contains a user-interface and some of the fundamental routines that are important in the finished software. A prototype with a user-interface provides the 'look and feel' of the product. Users can try it out and comment on what they like or don't like about the way it looks and how it appears to work.

One benefit of prototyping for a development team is being able to make changes to the finished product that incorporate the client's feedback. There is also a cost benefit, as producing a prototype is cheaper than producing the whole product.

It may not be possible with an algorithm design to actually show a formal prototype to clients. However, it is worth following the principle of developing a partial solution for a problem and then testing it before further refinements are made. You should not be reticent to make a 'first pass' attempt at a problem solution, knowing that some essential elements are not yet done. The 'prototype' algorithm will help to clarify some of the requirements and also allow the development of sections of the algorithm that can be usefully deployed in the final solution.

Thorough documentation

External documentation

Commercial software products are accompanied by documentation so that users can learn what the product does and what they need to do before using it. Users can also refer to the documentation whenever they need to clear up misunderstandings about the operation of the software. Such documentation is provided in print form separately to the software and is called 'external documentation'. Your algorithms will probably not need external documentation unless your teacher asks for it explicitly for assessment purposes.

Internal documentation

Internal documentation is written within the algorithm itself and is provided in two forms. First, a written block of comments can appear before the actual algorithm. This is used to introduce the reader to the algorithm and includes a brief description of its purpose, the author's name, the date of modification and any important assumptions that the writer has made. It is really a short technical and factual list of comments about the algorithm. The comments don't make the algorithm itself work any better; they just provide some details about it for the writer and others who might need to use it. Placing a comment block at the top of an algorithm is part of the culture of programming.

The second form of internal documentation consists of single lines of comments that are embedded in the algorithm at appropriate places. Such comments are used to add value and readability to complex instructions or to break up a long sequence of instructions into paragraphs. You should use these 'one-liners' whenever you feel they are needed. Don't use too many; if you do, your algorithm will become harder to read. These embedded one-line comments should not be used to describe instructions that are already easy to follow. In the sample algorithms in this chapter the double-slash notation has been used to indicate comments, like this:

```
// this is a comment!
```

You can adopt this notation if you wish. It comes from the C language and is also used in C++ and Java. If you don't like the notation, you can make up your own or use any of the other common styles, such as

```
# this is a shell script comment
; this is an Omnimark comment
REM this is a BASIC comment
{this is a Pascal comment }
[this is someone else's form of comment]
```

Listing assumptions

One crucial inclusion in the block of introductory comments at the top of a complete algorithm is a list of assumptions. You will need to decide on these so that readers of your algorithm can see what features it supports and what it purposely omits.

For example, you may have an algorithm that accepts input data values from a user. Your algorithm may not include checks that the user's data is legal or valid. This is an appropriate reason to include an assumption. You would state in your comments that the algorithm assumes that the user's input will be valid. If you are reading data from a file (dealt with later in the chapter), you may say that you assume that the file exists and/or that you assume that the file is not empty.

Stating assumptions allows you to set aside some difficult or unpredictable events and thus to avoid dealing with these in your design, but be careful not to assume away any fundamental requirements given in a problem.

Exercise 4.2

1 What strategy would you use to complete a small jigsaw puzzle? Would you use the same strategy to solve a large jigsaw puzzle? Why? Why not?
2 What does 'the scope of a variable' mean?
3 If a variable is created in a main module, it is also known in all of the sub-modules that the main module calls. True or false?
4 If a variable can be used only in the module that creates it, how can the value of a variable be made available in other modules?
5 A sub-module (or function) is made up of a name, arguments (or parameters), a set of instructions and a return value. How do these match the inputs, outputs and processes of a module?
6 What kind of variables are known to all of the modules in an algorithm?
7 Explain (or discuss) the benefits and disadvantages of having variables that are available to all modules in an algorithm.
8 What kinds of values should be chosen as sample data when testing an algorithm?

9 The following fragment contains a simple binary selection. How many different values for the variable 'speed' should be chosen in order to correctly test the fragment?

```
IF speed <= 100
THEN
    print 'Within the speed limit'
ELSE
    print 'Outside the speed limit'
ENDIF
```

10 Which values are appropriate for testing the above fragment, and which values are not appropriate?

11 What is a trace table? How can it aid the testing of an algorithm?

12 List two kinds of testing that can be used for a new software product.

13 What is a prototype?

14 How does developing a prototype help in the software development process?

15 List two kinds of documentation that should be employed in software development. Identify the separate uses of the two types of documentation.

Modules and functions

A function or module is a separate block of instructions that performs a well-defined task. A well-written function has clear input and output specifications.

Functions are the building blocks of algorithms and functional decomposition is the heart of structured programming. Of all the issues within algorithm design, decomposition is the most important. So you must be clear on how functions work and why they ought to be used. This section deals with these issues. It is modestly technical in nature. Concentration and practice are required if you want to become fully aware of the details.

General issues

Decomposition and reusability

You already know that functions are used to break an algorithm into separate parts so that the work done by the algorithm is distributed into smaller chunks. The other benefit is that once they are developed and tested, and if written the correct way and with a clear purpose, functions can be 'plugged into' any algorithm that requires their tasks. Reusability is important to software developers who want to be able to create robust products with a minimum of coding and in a minimum of time.

Function libraries

In the software development industry, large collections of well-tested and reusable functions are stored in 'function libraries'. Suitable function libraries can be included, or 'imported', into any new project. This means that the project can be built by designing the overall structure and then activating functions in the library to do much of the processing. Most programming languages are supported by 'standard function libraries'. These are fully refined and tested collections of functions that have been standardised by either the vendor of the language or an international body of experts.

The call and return mechanism

In all programming languages and algorithm designs, functions are activated by 'calling' them; the algorithm that activates a function is known as the 'calling code' or sometimes the 'client code'. When a function is called from somewhere in an algorithm, the current processing of that algorithm is postponed. The called function then starts working and takes control of all processing. When the function has performed its task and eventually terminates, control reverts to the algorithm that called it, which then continues on with any of its remaining instructions. So, in general terms, the principle of call and return is that, whenever a function terminates, control is always returned to the place from which it was called.

An analogy from ordinary life can be used to clarify this important principle. Suppose you are mowing your lawn. When you are about halfway through the mowing job, you feel the need for a cool drink, so you stop mowing, switch off the mower temporarily and go into your home to get a drink. You pour your drink into a glass and start to sip it. The telephone rings, so you put down the glass of drink and answer the telephone. Figure 4.2 represents the event up to this stage.

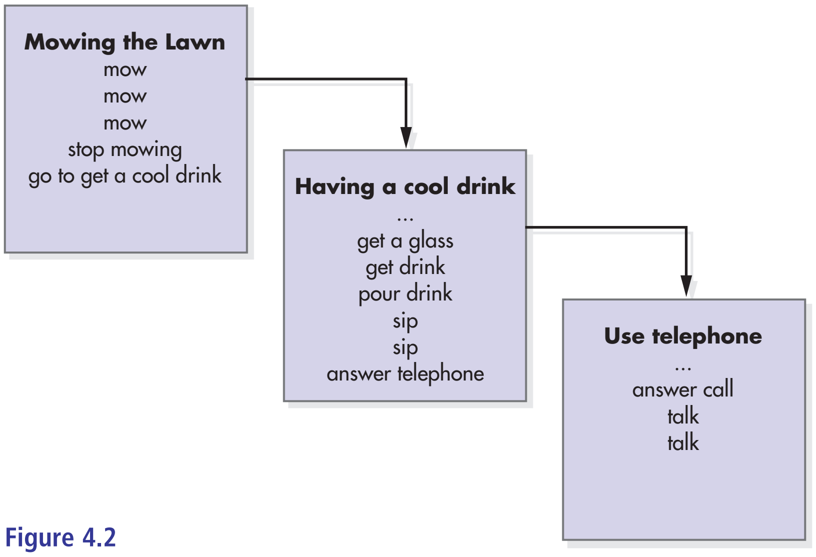


Figure 4.2

The situation at this instant is that you are involved in a telephone conversation—this is your current task. The drinking of the cool drink has been stopped or postponed and you have not yet finished the drink, but you do intend to finish it when you get the chance. Of course, the mowing task was also stopped and is only about half done. You intend to finish that job too when you get a chance.

Getting the cool drink is like a sub-module or function that has been called halfway through the mowing process. The mowing is the calling code. The telephone task is like a function that has been called halfway through the drinking module, so the drinking module is the caller of the telephone module.

To make this situation a simulation of an algorithm, you need to think about what will happen when the telephone call ends. When the call ends, you should go back and continue the task you were doing before the phone call started, that is, drinking. When you finish the cool drink, that function will be terminated. What will you do then? You should return to and complete the task you were doing before the drinking function, that is, mowing the lawn. Notice that, if you behave logically, when you finish any function you will always return to processing the code that activated the called function. This is what an algorithm or a computer program does. Figure 4.3 shows the return routes for the mowing, drinking and talking analogy. You should start looking at it from the beginning of the mowing activity.

Function anatomy

A correctly written function can have up to four parts: its name, its input, its output and a block of instructions. Of these, the name and the block of instructions are compulsory for all functions. The following sample pseudocode shows all four parts, the complete anatomy, of a typical function. It will be referred to in the following topics.

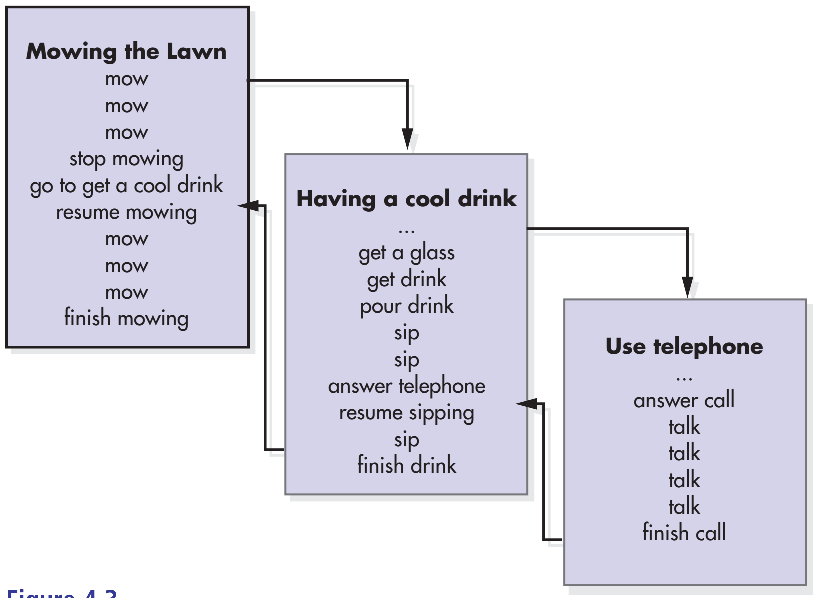


Figure 4.3

```
BEGIN findBiggest with firstNum and secondNum and thirdNum
    set bigNum to firstNum
    IF secondNum > bigNum
    THEN
        set bigNum to secondNum
    ENDIF
    IF thirdNum > bigNum
    THEN
        set bigNum to thirdNum
    ENDIF
    return bigNum
END
```

Function names

All functions must have names. The name of the function should be reasonably short and be readable and meaningful. The role of the name is to indicate what the function does. The name 'findBiggest' used in the sample above is an acceptable name and tells the reader that its purpose is to find or calculate the biggest value of some given values.

You already know about variable names, and function names follow similar guidelines. Function names are often made up of two or three words joined together to form a single name. There is no formal rule that tells you what is a good name and what is not. You just make up the names of your own functions so that they are readable and meaningful.

Function input

Input to a function is the data that is passed to it by the calling code. You saw earlier that values can be passed to a function when it is called. You will recall that this is the way that one module or function sends information to another.

Here is a fragment of an algorithm that calls the `findBiggest` function.

```
get score1 from the user
get score2 from the user
read highScore from the best player file
set winner to findBiggest with score1 and score2 and highScore
...
```

In this fragment the user might enter 456 into `score1` and 234 into `score2`. Then 345 might be read from a file into the variable `highScore`. When the function is called, the values 456, 234 and 345 are passed to the `findBiggest` function, just as if the call in the fragment above was really

```
findBiggest with 234 and 456 and 345
```

The three numbers are considered to be input to the function. In programming languages the values that are sent to functions are called 'arguments' or 'parameters'. You can use these names if you wish when referring to the values that any function accepts as input.

Function output

If functions can accept input via arguments, they must have some way to deliver output back to the calling code. In algorithms, as in most programming languages, we output information from a function by 'returning' it.

In the sample function above, the line

```
  return bigNum
```

shows what is being returned to the calling code. If we use the three values given in the last topic, 456, 234 and 345, as the sample input for this function, we can follow the logic of the function's instructions and see what value is returned. This can be done with a simple trace table, and you should verify yourself that the value returned is indeed 456, the largest of the three input values. You should desk check the function with other input values to make sure that you understand how it works.

It is natural in most cases for functions to have their return command as their very last instruction (as seen in the sample 'findBiggest' function). Some programmers actually design all their functions so that this is the case. However, it is not wrong to use the return command before the end of the instructions in a function as long as you realise that, whenever a return command is activated, the function is terminated at that instant and any instructions following the return command are ignored.

This is a subtle and technical aspect of module design and is a somewhat advanced concept. The example below shows a function whose purpose is to accept a month number, from 1 to 12 inclusive, and to send back the number of days in that month. For example, if we call the function with the month number 6, we expect to get back 30, since there are 30 days in the month of June.

```
BEGIN daysInMonth with monthNumber
    IF monthNumber < 1   OR   monthNumber > 12   // (1)
    THEN
        return 0  // (2)
    ENDIF
    CASEWHERE monthNumber IS    // (3)
        1, return 31
        // etc for other months
    ENDCASE
END function
```

This example function first checks that its input, the `monthNumber`, is valid (that is, that it is between 1 and 12 inclusive). If this is not the case, it is impossible to calculate the number of days. The technique the function uses to deal with this situation is to check for month numbers that are not valid, using the selection test on line (1) above. If this test is true, the function returns the value zero indicating that no days can be given for that month. What is subtle here is that, if the return command on line (2) is activated, the whole function terminates at that instant—at line (2). In this case, the multiple selection that begins on line (3) is never executed. Conversely, if the error test on line (1) is false, the function does not activate the return command on line (2); instead it continues with its processing from line (3) onwards.

The technique of using an early return command to terminate a function is called 'short circuiting' the function. The short-circuit technique is often used when error checking, as shown in the `daysInMonth` example above.

Function processing

It is reasonable to say that any function or module should do something. What it does is embedded between the keywords 'BEGIN' and 'END' and is often called the 'body' of the function.

Local variables

You will recall that variables can be created inside functions. These 'local' variables are created as they are needed, to assist with any calculations or other processing. There is no conflict between local variables of one function and those of any other function. The values are not shared across functions and a designer can create local variables in many functions using the same variable name for all of them.

In the 'findBiggest' function above, the variable 'bigNum' is a local variable. It is used to keep track of which of the input values is the biggest while the processing is taking place.

Example using function input and output

The aim of this example is to reinforce the principles explained above: that functions can accept input from the algorithm that calls them, that they can return information back to the calling algorithm, and that they can create and use local variables to make the local processing easier.

PROBLEM STATEMENT

The problem solved by this example concerns the conversion of temperatures from the old Fahrenheit scale (still used in the United States) to the new Celsius scale (as used in Australia and Europe). The main module of the algorithm is a simple user interface that requests the user to enter a temperature and also to indicate if it is Fahrenheit or Celsius. When the temperature is entered, the algorithm calls the appropriate sub-module (function) to convert the value to the other scale.

SAMPLE SOLUTION

Here is the main module of a solution algorithm.

```
BEGIN temperature conversion (main-module)
    print 'Please enter a temperature value'
    get degrees from user
    print 'Is this value in Fahrenheit (F) or Celsius (C)?'
    get scaleCode from user
    IF scaleCode = F
    THEN
        set resultDegrees to convertFtoC with degrees
        print 'the Celsius equivalent is' resultDegrees
    ELSE
        IF scaleCode = C
        THEN
            set resultDegrees to convertCtoF with degrees
            print 'the Fahrenheit equivalent is' resultDegrees
        ELSE
            print 'Error, can't decide which conversion to do'
        ENDIF
    ENDIF
END
```

CALLING A FUNCTION AND PASSING A VALUE

In a preliminary desk test of this algorithm we could assume that the user enters the number 110 into the variable 'degrees' and that the user specifies the scale to be Fahrenheit by entering the letter F into the `scaleCode` variable. With this sample data the algorithm above activates or 'calls' the function `convertFtoC` and passes the value 110 to it, as shown in Figure 4.4.

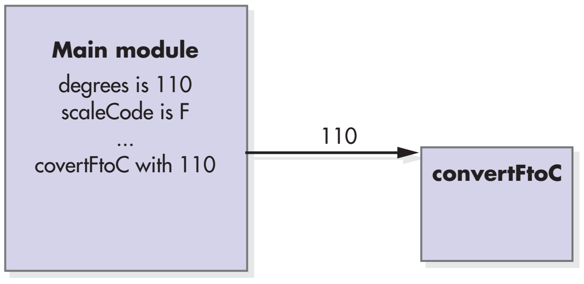


Figure 4.4

PROCESSING IN THE FUNCTION

The function `convertFtoC` accepts the value 110 as input, converts that value to its equivalent Celsius value and then returns the result back to the main module. Here is the pseudocode for the `convertFtoC` function. Note that its input is available to it in the argument or parameter 'degrees'.

```
BEGIN convertFtoC with degrees
    set answer to (degrees – 32) × 5 / 9
    return answer
END
```

If the argument 'degrees' comes into this function with the value 110, the arithmetic on the first instruction calculates the local variable 'answer' as (110 – 32), which is 78, multiplied by 5, which is 390, divided by 9, which gives 43.3.

FUNCTION OUTPUT

The value 43.3 is currently in the local variable 'result'. The return command sends this value back to the main module, as shown in Figure 4.5.

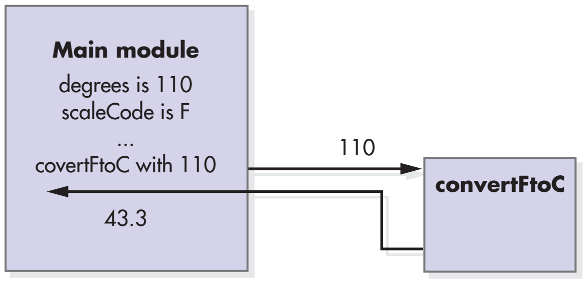


Figure 4.5

ACCEPTING DATA FROM A FUNCTION

Note again how the `convertFtoC` module was called in from the main module:

```
set resultDegrees to convertFtoC with degrees
print 'the Celsius equivalent is ' resultDegrees
```

So the value 43.3 which comes back from the function is captured into the variable 'resultDegrees' and is then, on the next line, printed onto some output device.

FOR YOU TO DO

The techniques used above are typical of how functions are used; data is passed to them, some processing takes place, and then data is returned from them to the calling algorithm. The other module `convertCtoF` can be written and traced in exactly the same way. Complete it, desk check it, and draw diagrams similar to those above showing how it works. The formula for converting from Celsius to Fahrenheit is
Fahrenheit = 9 × Celsius / 5 + 32

Boolean functions

A Boolean value is either true or false. A function that returns either true or false is called a 'Boolean function'. Boolean functions are very popular in all programming languages and so are relevant to a study of algorithms. These functions are sometimes called 'query functions' because they usually perform a test on the value of their input. They also often have a name starting with the word 'is', as seen in the example below.

```
BEGIN isVoter with age
    IF age < 18 OR age > 80
    THEN
        return false
    ELSE
        return true
    ENDIF
END
```

This function accepts a single argument, a value that represents a person's age; if that is not a legal voting age, the function outputs `false`, otherwise it outputs `true`. To see why Boolean functions are often named with an 'is' prefix, look at the following fragment of an algorithm that calls the `isVoter` function.

```
BEGIN
    read name and age
    IF isVoter with age // (1)
    THEN
        print name 'can vote'
    ELSE
        print 'Sorry' name ', you can't vote.'
    ENDIF
END
```

The instruction on line (1), where the function is called, reads almost like part of an English sentence: 'If a person of this age is a voter …'

Exercise 4.3

1. What is the difference between an arbitrary collection of functions and a function library?
2. Why is the code that calls a function sometimes called 'client code'?
3. When a function returns, where does it return to?
4. A function (or module) can be considered as a small independent algorithm. How is the input, the output and the processing of an algorithm applied to a function within an algorithm?

5 The instruction ‘return’, used in a function, has two purposes. One is to send a value back to the function’s calling code. What is the other purpose?

6 Does a return instruction have to be the last instruction in a function?

7 What is the meaning of the term 'short circuit' with respect to functions?

8 Complete the function 'convertFtoC' which was described in this section.

9 Study the following algorithm and draw a diagram to describe it. The diagram should indicate with arrows how values are passed to the ‘yearsToDays’ function and how values are returned from it.

```
BEGIN
    get ageInYears from the user
    set days to yearsToDays with ageInYears
    print days
END
BEGIN yearsToDays with age
    set result to age × 365.25
    return result
END
```

10 What is a Boolean function?

11 Why do Boolean functions often have names starting with the word ‘is’?

12 Why are Boolean functions popular in all programming languages?

13 Why are Boolean functions so called? You may have to do some research to answer this question. Start your research by looking for references to George Boole.

Programming problem

14 A sports club has a membership policy that favours young people. Any person who is under 21 years of age is allowed to join the club. Only 10 members over 21 are allowed and they are called senior members. If more than 10 applications are received from people over 21, the extra people are placed on a waiting list.

Write an algorithm that allows the user to enter the age of as many people as the user wishes, with each entry of an age being an application to join the club. As each age is entered, the algorithm is to print a message that indicates whether the application is accepted as an ordinary member, is accepted as a senior member, or is put on the waiting list.

When the user enters a special age of 0, the algorithm is to terminate after displaying the number of ordinary members, the number of senior members and the number of people on the waiting list.

Write your algorithm with a single main module supported by several separate functions.

Arrays: basic concepts

This section deals with the array data structure. It reviews the fundamental concept of an array and outlines some standard techniques for using arrays.

The array concept

An array consists of a collection of data and is officially called a ‘data structure’. The array data structure holds several individual values, each in a separate ‘array cell’. The cells in an array are arranged in a linear structure in a similar way to rooms in a hotel corridor. Just as rooms in a hotel are numbered, so the cells in an array are numbered or ‘indexed’, each cell having its own index.

Naming and creating arrays

When algorithms are written, variables are used to store information, with each variable referring to just one data value. An array variable can be used to name a collection of different values. In an algorithm that deals with just one person's name, a variable might be created to hold the name, like this:

```
set personName to 'Susan'
```

If the algorithm needs to work with several people's names, one variable can be created for each, like this:

```
set name1 to 'Susan'
set name2 to 'Lee'
set name3 to 'Kerry'
set name4 to 'Stephen'
```

Creating a separate variable for each of many names means that an algorithm becomes long and working with all the separate variables becomes tedious. This is where an array structure can be very convenient. To store the four names above in a single variable, an array structure can be created in an algorithm as follows:

```
nameList is an array of names indexed from 1 to 4
```

This creates an array called 'nameList'. It specifies explicitly what kind of data the array will hold, how many values it will hold, and how the values will be indexed or numbered. It is useful to visualise this array. Figure 4.6 shows the basic structure.

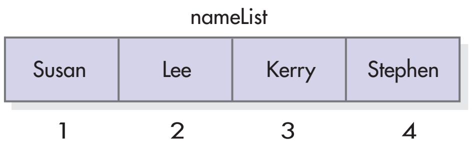


Figure 4.6

What is important to realise here is that the variable 'nameList' refers to all of the people's names and each individual name is located in a separate cell (or 'element') of the nameList array. Each cell is numbered or indexed and in this example the simple numbers 1 to 4 have been used as the indices. It is easy to access an individual person's name by using the index of their particular cell. The following algorithm fragment displays one of the names—the one in cell number 2.

```
print nameList[2]
```

Arrays hold a single data type

Arrays can hold many values but all of the values must be of the same type. This means that you must be clear about what kind of information you wish to use before you create an array to hold it. If you want to keep track of the share price of 20 different companies, a suitable array might be:

```
sharePrice is an array of real numbers indexed from 1 to 20 // (1)
```

Once created, it would be easy to store a price into this array. Suppose that the company 'Magic Corp' has a share price of $2.89. The price could be stored into the fifth cell, like this:

```
set sharePrice[5] to 2.89
```

It would be impossible to store the name of the company in this array:

```
set sharePrice[5] to 'Magic Corp' // WRONG and ILLEGAL!
```

The `sharePrice array` has been defined, in line (1) above, to explicitly hold real numbers, not company names. To hold the names of the companies a different array variable would need to be created.

Arrays are of fixed length

The size or length of an array, the number of cells in it, must be specified when the array is created. The important implication of this rule is that a writer needs to know or to estimate how many cells are needed at the time the algorithm is written. The algorithm cannot add extra cells dynamically while it is being executed; nor can it make an array shorter by removing cells.

It is not necessary to use all the cells of an array. You can store data into an array and legally have some cells with nothing in them. Of course it is pointless to create arrays which are so long that the majority of their cells are unused; so you usually need to make a reasonable guess when deciding on an array length, often with a few extra cells allocated for safety.

Indexing arrays

The cells in every array are numbered or indexed. Any sequence of whole numbers can be used to index an array and it is sometimes convenient to use indices that have some meaningful value. If an algorithm needs to deal with the total rainfall values for all the years of the last decade, an array could be created as follows:

```
rainfall is an array of whole numbers indexed from 1 to 10
```

If there were 670 mm of rainfall in 1991, this value could be stored in the array like this:

```
set rainfall[1] to 670
```

In this case it might be better to index the array with the actual year numbers, as:

```
rainfall is an array of whole numbers indexed from 1991 to 2000
```

Thus 1991's rainfall could be stored as:

```
set rainfall[1991] to 670
```

Indexing the rainfall array in this way adds semantic value to your algorithm.

Several currently popular programming languages, such as C, C++ and Java, do not allow such indexing. In these languages every array must be indexed so the first cell is number 0 (zero). So, for the rainfall example above, these languages would need an array similar to our pseudocode

```
rainfall is an array of whole numbers indexed from 0 to 9
```

and 1991's rainfall of 670 mm would be stored as

```
set rainfall[0] to 670
```

Traversing arrays

Arrays are used to hold and manage many values. For many problems algorithms require every value in an array to be processed; this can be done by using a loop to 'iterate over' the entire array, processing each value as its cell is visited. This is called 'traversing the array'. The sample module below accepts an array that holds whole numbers. The module traverses the array and adds all the values together into a total; it assumes that the array 'numberList' has been created in the calling code and is already full of numbers. A counter-controlled 'while' loop is used.

```
BEGIN sumArray with numberList
    set total to zero                          // (1)
    set counter to 1                           // (2)
    WHILE counter <= last index of numberList  // (3)
        add numberList[counter] to total       // (4)
        increment counter
    ENDWHILE
    return total
END
```

This module could be called from an algorithm fragment as follows:

```
set result to sumArray with numberList
print result
```

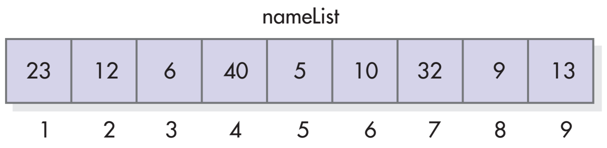


Figure 4.7

If the array numberList is as shown in Figure 4.7, then the fragment would print 150.

The sumArray function accepts the array numberList as an argument. It creates a local variable 'total' which is initially set to zero (line 1), and then creates another local variable 'counter' which is initially set to 1 (line 2). The condition on the 'while' loop at line 3 checks that the value of the counter is less than or equal to the last index of the array. This is certainly true initially as the counter is 1 and the last index of the array is 9. Since the condition is true, the 'while' loop performs its actions which add the current value in the array onto the total (line 4) and bump the counter up to its next value.

The 'while' loop will continue to iterate and thus traverse the whole array adding the value of each array cell into the total. When the counter reaches 10, the loop's condition will fail (you should definitely check this yourself) and the loop will terminate. The final line in the function sends the total back to the calling algorithm where it gets printed.

Alternatively, this array traversal can be completed with a fixed loop, as shown in the version of the sumArray function below. A fixed loop, using the pseudocode keyword 'FOR', is often used in array traversals.

```
BEGIN sumArray with numberList
    set total to zero
    FOR counter goes from 1 to last index of numberList
        add numberList[counter] to total
    ENDFOR
    return total
END
```

Sample functions

The following sample functions show some other traversals of arrays. All the samples work on the array numberList shown in Figure 4.7.

It is recommended that you study each function closely and that you use a desk check and a trace table to check that each function works correctly. Use the diagram in Figure 4.7 as the input array.

Finding the largest value in an array

This function traverses the array and returns the largest value in it. The function initially stores the first value in the array into a local variable 'largest' and then iterates over all the other cells, adjusting the value of the variable 'largest' whenever necessary. An inspection of Figure 4.7 reveals that the value returned by this function is 40. Your desk check should confirm this.

```
BEGIN findLargest with numberList
    set largest to numberList[1]
    FOR counter goes from 2 to last index of numberList
        IF numberList[counter] > largest
        THEN
            set largest to numberList[counter]
        ENDIF
    ENDFOR
    return largest
END
```

Finding the position of the largest value in an array

This is similar to the previous `findLargest` function. However, instead of returning the largest value in the array, it returns the index of the largest value. An inspection of Figure 4.7 reveals that the value returned by this function is 4. You should be able to confirm this with your desk checking.

```
BEGIN findLargestPosition with numberList
    set bigPosition to 1
    set largest to numberList[bigPosition]
    FOR counter goes from 2 to last index of numberList
        IF numberList[counter] > largest
        THEN
            set largest to numberList[counter]
            set bigPosition to counter
        ENDIF
    ENDFOR
    return bigPosition
END
```

Simple linear search of an array

Advanced array processing, including searching and sorting, is dealt with later in the chapter. The following function provides a simple linear search only and you should be able to follow it using similar desk-checking techniques to those you have used in the previous examples.

This function accepts the array and a value to search for. It traverses the array, checking each cell's value against the search value. As soon as it locates the first cell value that is equal to the search value, it returns the index of the cell. In cases where the search value cannot be found anywhere in the array, the function returns –1.

```
BEGIN linearSearch with numberList and searchValue
    set counter to 1
    WHILE counter <= last index of numberList
        IF numberList[counter] = searchValue
        THEN
            return counter  // (1)
        ENDIF
        increment counter
    ENDWHILE
    return –1
END
```

Note carefully that this function will return the index of the `searchValue` as soon as it is found. The return instruction (on line 1) will terminate the function immediately when this happens. This is an example of the 'short-circuit' technique discussed earlier.

If the above function is called as given in the fragment

```
 set position to linearSearch with numberList and 10
 print position
```

it will print 5 since this is the index of the cell containing the value 10.

If it is called with the fragment

```
 set position to linearSearch with numberList and 17
 print position
```

it will print –1 since the number 17 cannot be found in the array.

You are advised to check both results with your own desk check. You should also do the exercises to gain experience with other similar array traversals.

Exercise 4.4

1 Is it possible to store people's names and also their ages in the same array?

2 An array is declared as follows:

```
 someList is an array of whole numbers indexed from 1 to 10
```

Which of the following statements are legal?

a `someList = 45`

b `someList[11] = 45`

c `someList[10] = 3.2`

d `someList[1..10] = 0`

e `someList[−1] = 34`

3 Using the declared array above, what is printed by these statements?

a `print length of someList`

b `print last index of someList`

c `print first index of someList`

4 You have to store all the marks of the students in your class and you choose to use an array as the data structure. How big should the array be?

5 Your array from question 4 is used in an algorithm, and while the algorithm is running it is found that there are more students in the class than cells in the array. What can be done about this within the algorithm?

6 What does it mean to 'traverse' an array?

7 Write a function that returns the smallest value in an array of whole numbers.

8 Without modifying your function from question 7, desk check it using the values below as the values of the cells of the array:

```
23  56  12  56  22  1  45  67  77
```

9 A common mistake in writing a function to find the smallest value in an array is to wrongly assume that all the values are positive. Perform the same desk check using these values:

```
−3  −5  −78  −99  −33  −12  −2  0
```

If your solution does not work correctly, modify it.

10 Write a different version of your function from question 7 so that instead of returning the smallest value in the array it returns the index of the smallest value.

11 Write a function that accepts an array of real numbers as an argument and returns the average of the values in the array.

12 Write a function that accepts an array of real numbers and returns the number of cells that contain values that are bigger than the average cell value. Use your solution to question 11 as part of your solution.

13 Write a function that prints the values of all the cells in an array from the last cell to the first cell.

14 Write a function that prints all the values of the cells in an array with a comma between them; for example:

```
12, 45, 67, ..., 3, 21
```

Make sure that a comma is not printed after the last cell value nor before the first cell value.

15 Write a Boolean function that accepts an array of real numbers in one argument and a single real number in another argument. The function should return 'true' if any cell value is equal to the given number and 'false' otherwise. Name your function appropriately.

16 Write a function that accepts an array of real numbers in one argument and a single real number in another argument. The function should perform a linear search of the array and return the cell value that is closest in value to the given number.

17 Write a function that accepts an array of real numbers in one argument and a single real number in another argument. The function should perform a linear search of the array and return the cell index of the cell whose value is closest in value to the given number.

18 How efficient is a linear search? To answer this question, consider an array containing 100 cells when the correct search value is in the second cell. Now consider the situation in which the correct search value is in the 98th cell.

Advanced sorting and searching techniques

This section deals with some standard ways to search and sort collections of data in arrays. Although the techniques are well known in the software development field, they can be considered as advanced work in the context of any Higher School Certificate course. Searching is dealt with first, and follows on from the simple linear search covered in the previous section.

Searching arrays

Linear search

An example of a linear search of an array has been given in the previous section. You should note the logic of the linear search, which is to simply start at the beginning of the array and proceed along the array, checking each cell's value against some search value. As soon as the search value is found in one of the cells, that cell's index is output, indicating that the search value has been found and where it is in the array. If the search value cannot be found some special value must be output to indicate this. In the example function on p. 98, –1 is returned if the search is unsuccessful.

A linear search is easy to understand, but its efficiency is completely dependent on where the search value is in the array. If the value being searched for is near the beginning of the array, the linear search will locate it quickly. If the search value is near the end of the array, the linear search will need to traverse almost all the cells. This is obviously inefficient if the array is long.

Binary search

Searching an array can be done much more efficiently using a binary search, but this can only be done if the values in the array are already sorted in order. Later in this section you will be presented with several standard sorting techniques, but for now you should assume that there is an array of values that is already in sorted order. Such an array can be created with the algorithm instruction

```
numberList is an array of whole numbers indexed from 1 to 9
```

A diagram of this array, with sample values already assigned to the cells, is given in Figure 4.8.

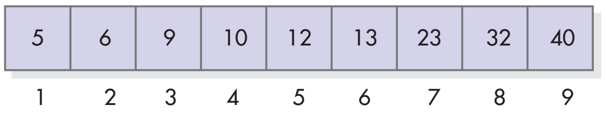


Figure 4.8

The binary search takes advantage of the sorted nature of the array and does not simply start searching at the start of the array like the linear search does. The binary search applies the same logic to the searching process that you would use if you were searching for someone's name in a telephone directory. For example, you know that a telephone directory is already sorted, so if you were looking for 'Jackson' you would start in the first half of the book, since the name 'Jackson' occurs in the first half of the alphabet. You might open the directory at some arbitrary place in the first half of the book and then check the name of the people on that page. If the page contained the name 'Greer', you would flick forward looking for 'Jackson'. If one of the names on the page was 'Patrius', you would flick backwards, because you know that 'Jackson' will occur before 'Patrius'.

The binary search algorithm employs similar logic, although it is not quite as clever as you would be at finding the correct search value. It always starts by checking the middle value of an array to see if that value matches the search value. If that middle value is not correct, the binary search then determines if the search value is in the section of the array before the middle value (the first half of the array) or after the middle value (the second half of the array). Once this is known, the binary search then applies the same logic to that particular half of the array that it used for the whole array. That is, it selects the new middle value, checks it against the search value and again divides the search area in half if the value if incorrect.

Before dealing with the actual algorithm in pseudocode, the logic of the binary search technique is presented here using diagrams.

Suppose that we have the array given in Figure 4.9 and we are searching for the value 10. That is, we want to know the index of the cell that contains 10. The first step, as outlined above, is to find the middle value of the array and check it. The end indices of the array are 1 and 9, so the middle cell will be number 5 (note that 5 is the average of 1 and 9). We check cell 5. Does it contain the search value? No, it contains the value 12. At this point, we can visualise the situation as shown in Figure 4.9.

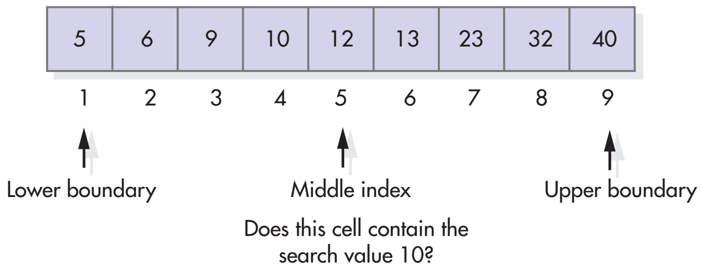


Figure 4.9

The binary search must now decide which half of the array to use for further searching. Since the value 12 which was found is greater than the search value, it is clear that the lower half of the array must contain the 10. The upper boundary of the search region is moved down to cell 4, the middle of the new region is calculated as cell 3 (the average of 1 and 4 rounded up), and this value is then checked. This situation is shown in Figure 4.10.

Once again the search fails to locate the correct value, as the cell being checked contains 9, not 10. However, this time the value 9 is less than the search value, so the lower boundary is moved up to cell 4. The situation is now that the upper boundary is at cell 4 and the lower boundary is also at cell 4. The middle of this range is obviously 4 (the average of 4 and 4) and the binary search now looks like Figure 4.11.

The search value has now been found, in cell 4, and the binary search routine stops and outputs the value 4.

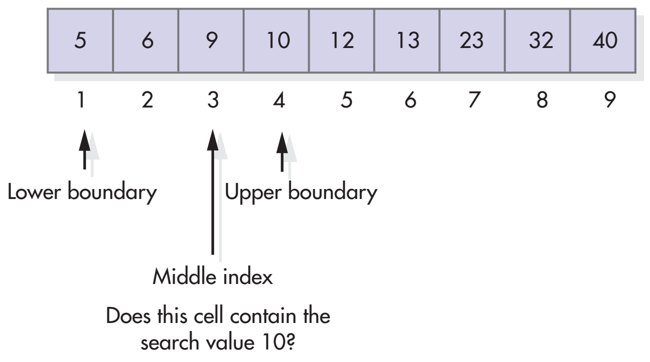


Figure 4.10

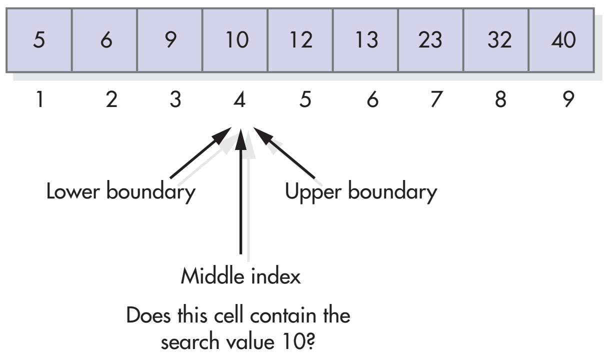


Figure 4.11

In situations where the search value is nowhere in the array, the binary search will finish with the upper boundary being less than the lower boundary. In this case, the routine needs to output some special value, such as –1, to indicate that the search has failed.

The algorithm for the binary search is given here as a function in pseudocode. It is quite complex compared with the linear search algorithm but exactly matches the logic used in the diagrams and discussion above.

```
BEGIN binarySearch with numberList and searchValue
    set lowBoundary to first index of numberList
    set highBoundary to last index of numberList
    WHILE highBoundary >= lowBoundary
        set middleIndex to (highBoundary + lowBoundary) / 2
        IF numberList[middleIndex] = searchValue
        THEN
            return middleIndex    // EUREKA, short circuit
        ENDIF
        IF numberList[middleIndex] > searchValue
        THEN
            set highBoundary to middleIndex – 1
        ELSE
            set lowBoundary to middleIndex + 1
        ENDIF
    ENDWHILE
    return –1
END
```

This function can now be plugged into any algorithm that requires it.

Example of binary search

This situation could be part of a larger application which a share trader might use to record and manage a portfolio of five different stocks. We set up the problem with two arrays, one of which contains the names of several companies in which the trader holds stock. This array would be created as

```
 companyList is an array of names indexed from 1 to 5
```

and is represented in Figure 4.12. This array is sorted into alphabetical order and so a binary search on it is possible.

Coles Myer	Davnet	Reckon	Secureld	Woolworths
1	2	3	4	5

Figure 4.12

An array of share prices in parallel to the `companyList` array is shown in Figure 4.13. A parallel array is one where the indices used for the cells of one array match those used in another array. So if the company name 'Davnet' is in cell 2 of the first array, its price is in cell 2 of the second array.

7.99	1.89	2.03	10.67	12.56
1	2	3	4	5

Figure 4.13

The following algorithm allows the user to enter the name of any company and, in response, prints the share price of that company. If the entered company name is not one of those in the portfolio, the algorithm prints a suitable message.

```
BEGIN find share price (main module)
   companyList is an array of names indexed from 1 to 5
   priceList is an array of real numbers indexed from 1 to 5
   get companyName from the user  //(1)
   set foundIndex to binarySearch with companyList and companyName
   IF foundIndex =–1
   THEN
      print 'No such company in portfolio'
   ELSE
      print 'For the company ' companyName '
      print 'the share price is ' priceList[foundIndex]
   ENDIF
END
```

A desk check of this algorithm can be done by making up a company name for input on line 1. If you use 'Davnet' as sample data, the variable 'foundIndex' should capture the value 2 when the `binarySearch` function is called. In this case the output from the algorithm will be:

```
 For the company Davnet
 the share price is 1.89
```

If the company name 'Telstra' is input, since that name cannot be found in the array the 'foundIndex' variable will capture –1, and the output will be:

```
 No such company in portfolio
```

You should perform several more desk checks of this algorithm, each time using a different company name. Make sure you trace the binary search function itself as well as the main module above. By doing this you will gain valuable practice at following the binary search logic.

Sorting arrays

There are several standard sorting techniques that can be applied to arrays. They are not simple. You will need to study the functions closely and desk check each one several times to ensure you understand it.

Bubble sort

This routine is popular among novice programmers and is reasonably efficient. Its main logical structure is based on traversing an array and switching adjacent pairs of values that are not in the correct order. After one traversal, the largest value will have 'bubbled' to the end of the array. This is repeated until all the values are in their correct cells, with the array completely sorted.

Figure 4.14 is an unsorted array of whole numbers, identical to the one used for the linear search earlier.

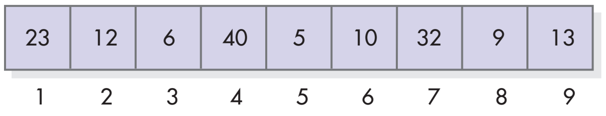


Figure 4.14

The bubble sort starts at the beginning of the array and compares the first two cell values. These are cell 1 containing 23 and cell 2 containing 12. These are not in sorted order so the values are swapped, as shown in Figure 4.15.

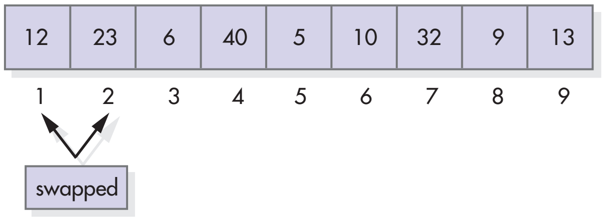


Figure 4.15

The next two cells (cell 2 and cell 3) are then similarly compared and, if necessary, their values swapped. Since 23 is larger than 6, the values do need swapping, and the result is shown in Figure 4.16.

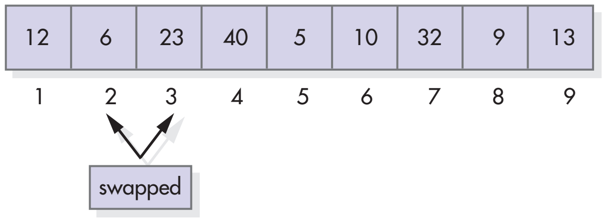


Figure 4.16

Next, the values in cell 3 and cell 4 are compared. These are already in the right order and so do not need swapping.

Continuing this activity, swapping adjacent values as necessary as the array is traversed, results in the largest number, 40, being moved into the last cell. Figure 4.17 shows this and also shows the changed positions of the other values after the traversal. Use a pencil and paper diagram yourself to fill in the swapping steps between Figure 4.16 and Figure 4.17, and then verify that the diagrams are correct.

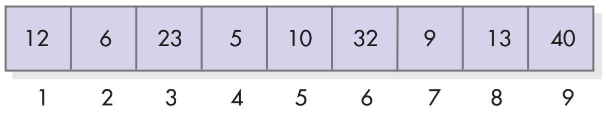


Figure 4.17

The swapping can easily be done in a function like this

```
BEGIN swap with value1 and value2
    set temp to value1
    set value1 to value2
    set value2 to temp
END
```

The complete traversal and its swapping can also be written as a function. The function accepts the array as an argument. The local variable 'cellNumber' is initially set to the second index of the array and the comparisons are done between the value at position `cellNumber` and the value before it. The function is called 'bubbleUp' because it has the effect of making the largest value in the array bubble up to the last cell.

```
BEGIN bubbleUp with someArray
    set cellNumber to first index of someArray + 1
    WHILE cellNumber <= last index of someArray
        IF someArray[cellNumber – 1] > someArray[cellNumber]
        THEN
            swap someArray[cellNumber – 1] and someArray[cell Number]
        ENDIF
        increment cellNumber
    ENDWHILE
END
```

So far we have only seen how to move one cell value, the largest one, into its correct position. To completely sort the array, this routine must be repeated once for each cell in the array. The following function, 'bubbleSort', does this.

```
BEGIN bubbleSort with someArray
    FOR counter goes from first index to last index of someArray
        bubbleUp with someArray
    ENDFOR
END
```

Using our sample array, given initially in Figure 4.14, the above function would call the sub-module 'bubbleUp' nine times, after which the entire array would be sorted. You definitely need to desk check this using pencil and paper diagrams and with close attention to tracing the values of the variables.

The bubble-sort routine presented here is effective but not completely optimised. The entire array can be sorted with fewer repeats of the 'bubbleUp' routine provided we can check each time that at least one value has been swapped. Try to develop such an optimisation. A possible strategy is to turn 'bubbleUp' into a boolean function, so it returns 'true' whenever a swap is done. The 'bubbleSort' function can then be modified as follows:

```
BEGIN optimisedBubbleSort with someArray
    REPEAT
        set aSwapWasDone to bubbleUp with someArray
    UNTIL NOT aSwapWasDone
END
```

In this form the bubble sort is quite efficient. It is particularly quick when the values in the array are only slightly out of order.

Selection sort

The logic of the selection sort is easy to follow, which makes it ideal for beginning programmers. The essential idea is that we place a marker at the first cell in the array and then search through the array from that position onwards looking for the smallest value. When the smallest value is found, it is swapped with the marked cell's value. This naturally places the smallest value at the front of the array. The next step is to move the marker to the next cell and repeat the process. When the marker reaches the last cell, the array is sorted.

The diagrams below help to explain this process. Figure 4.18 shows the unsorted array.

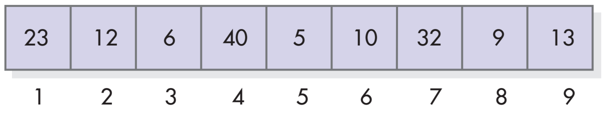


Figure 4.18

Start by marking the first cell and then look for the smallest value from that position on. Figure 4.19 shows this activity.

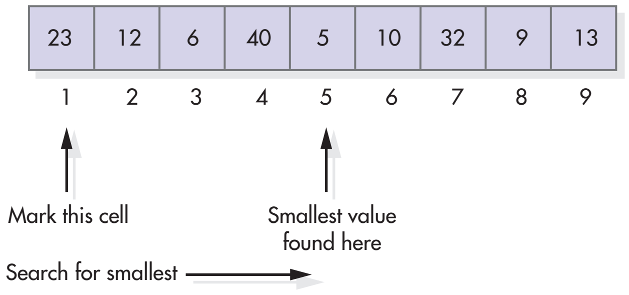


Figure 4.19

These two values are swapped to produce Figure 4.20.

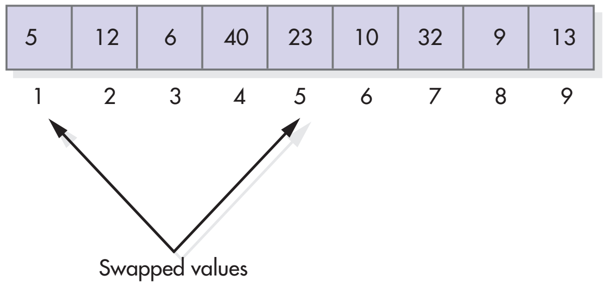


Figure 4.20

Now the first cell contains 5, the smallest value in the entire array. Mark the next cell, then find the smallest from that position onwards, as shown in Figure 4.21.

These two values are then swapped and the first two cells are in sorted order.

Draw the diagram for yourself and adjust it by repeating the 'mark next cell, then find smallest, then swap' routine. If you do this, you can verify that the selection sort works correctly.

The algorithm function below locates the smallest cell in an array by searching from a given position in the array. It will be used later in the complete selection-sort algorithm.

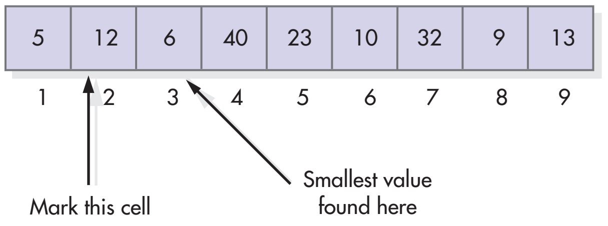


Figure 4.21

Notice that when this function is called, the index of the starting point (`markedCell`) and the name of the array (`numberList`) must be passed to it.

```
BEGIN getSmallest with markedCell and numberList
set smallIndex to markedCell   // (1)
    set smallValue to numberList[smallIndex]      // (2)
    FOR counter goes from markedCell to last index of numberList
        IF numberList[counter] < smallValue       // (3)
        THEN
            set smallIndex to counter
            set smallValue to numberList[counter]
        ENDIF
    ENDFOR
    return smallIndex
END
```

Exercise 4.5

1 A binary search of an array can be done correctly only when the cells are in sorted order. Why is this so?

2 How efficient is a binary search? In answering the question consider a sorted array containing 100 cells with the correct search value located in cell 2. Also consider the situation when the correct search value is in cell 98.

3 A simple guessing game can be played by two people. One person (the owner) makes up a secret whole number in the range 1 to 1000 (inclusive). The other person (the guesser) attempts to guess the number. Each time a guess is made, the owner says ‘Too big’ if the guess is larger than the secret number, ‘Too small’ if the guess is smaller than the secret number or ‘Correct’ if the guess is right. Play the game several times with a friend, taking turns as the guesser and the owner.

4 Using the game in question 3 as a guide, what is the minimum number of guesses that are needed to identify the secret number? What is the maximum number of guesses?

5 If the game in question 3 is played with a secret number in the range 1 to 100 inclusive, what is the minimum number of guesses required to ensure that the number is guessed correctly?

6 What is the minimum number of guesses required to ensure a correct identification of the secret number if the range 1 to 10 000 is used?

7 Study the bubble sort presented in this section. Desk check the bubble sort on an array of whole numbers containing the four values 23, 56, 12 and 6.

8 Desk check the bubble sort on an array containing the four values 5, 19, 45 and 78. Note that the values are already in sorted order

9 Desk check the bubble sort on an array containing the values 78, 45, 19 and 5. Note that the values are in reverse sorted order.

10 For questions 7, 8 and 9, count how many swaps were done during the bubble sort.

11 Desk check the selection-sort algorithm shown in this section on the value sets in questions 7, 8 and 9.

12 For each of the desk checks done on the value sets in question 11 count how many times the values in the array were swapped.

String processing

The word 'string' means 'a string of characters', one character after the other, forming a sequence. The word is used in the same sense as 'a string of pearls'. It refers to a sequence of characters that form a word, a sentence, a name or a message. Computer programs need to be able to deal with strings because they are used very often in most applications. A quick look at the user-interface of any software package will verify this: screens are full of words and messages, in menus, help systems, error messages and so on. There are also many programs that must deal directly with strings because their core purpose is processing words; word-processing packages, email clients, publishing systems and so on all require fast and sophisticated routines to process strings efficiently. Some of these routines are examined in this section.

String concepts

In traditional systems strings were not defined directly in a programming language. Instead, a programmer had to work with the individual characters that made up a message or word. To do this efficiently, the programmer used techniques which were very similar to those seen in the previous two sections. That is, a string was represented as an array of characters.

It is a tedious job for programmers to write all their own character array processing routines from scratch. Some standardised functions are required, and once these are written and well tested they can be stored in a function library and used over and over again in future applications.

Let us assume at this stage that no such library is available. One often-used, low-level, string-processing function is written to verify that strings are indeed just a sequence of characters. A number of similar functions are then discussed which are considered as 'given' or 'standard'. The section concludes by looking at several useful and interesting routines that extend the standard ones.

A low-level string function: concatenation of two strings

'Concatenate' is a fancy word that just means 'to join together'. A common task in programming is to join two words together to form a single result. A typical example occurs with people's names. If we have someone's first name, such as 'Homer', in one array of characters and their second name, 'Simpson', in another, we may wish to join these to form a single string containing 'Homer Simpson'.

The following function does the work of concatenating two such arrays of characters.

```
BEGIN concatenate with string1 and string2
    set len1 to length of string1
    set len2 to length of string2
    bigString is an array of characters indexed from 1 to len1+len2

    FOR counter goes from 1 to len1
        bigString[counter] = string1[counter]
    ENDFOR
    FOR counter goes from 1 to len2
        bigString[len1+ counter] = string1[counter]
    ENDFOR
    return bigString
END
```

The first three lines of this function find out the length of the two strings and create a single string big enough to accommodate all the characters (`bigString`). The first loop copies each of the characters from the first string into the `bigString` using the loop counter for indexing.

The second loop is similar, but the loop counter is used for indexing in a slightly different way. In this case, the position in `bigString` where each character goes is after the existing characters. The strings containing the separate parts of a name are shown in Figure 4.22.

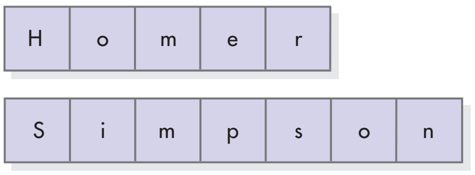


Figure 4.22

These have been defined as characters arrays as:

```
firstName is an array of characters indexed from 1 to 5
secondName is an array of characters indexed from 1 to 7
```

We could join them together with a call to the concatenation function like this:

```
set fullName to concatenate with firstName and secondName
print fullName
```

Of course, the output will be:

```
HomerSimpson
```

which is technically correct, even though it might not be exactly what we want when displaying a full name.

The fact that there is no space between the two original strings in the concatenated version is not a fault with the function. If we modified the function so that a space was included we would, in fact, be making a grave mistake. A well-defined, technically correct and general-purpose function that does exactly as it is supposed to is more useful than a specialised one. It is a building block on which specialised functions can be based if necessary.

To extend our concatenation function, consider a situation where several separate words have to be joined to become a sentence. Assume that we have four words, defined in strings as word1, word2, word3 and word4. To join them into a sentence we need to concatenate them, with a space between each but no space at the end. The following simple algorithm does this.

```
BEGIN make a sentence
    space is an array of characters indexed from 1 to 1
    set space[1] to ' '
    set sentence to concatenate with word1 and space
    set sentence to concatenate with sentence and word2
    set sentence to concatenate with sentence and space
    set sentence to concatenate with sentence and word3
    set sentence to concatenate with sentence and space
    set sentence to concatenate with sentence and word4
    display sentence
END
```

A more general solution can be applied to a list of words that are held in an array (an array of strings). A function that accepts an array of words and delivers a single string containing all of them concatenated, with a space between each, is as follows. It uses exactly the same logic as the algorithm above but in a more generalised way.

```
BEGIN makeParagraph with wordList
    space is an array of characters indexed from 1 to 1
    set space[1] to ' '

    set paragraph to wordList[1]
    FOR counter goes from 2 to last index of wordList
       set paragraph to concatenate with paragraph and space
       set paragraph to concatenate with paragraph and wordList[counter]
    ENDFOR
    return paragraph
END
```

Notice that there is no dangling space at the end of a paragraph created with the above function. Can you see why?

We can consider the concatenate function above as a low-level function because it deals directly with the underlying array which contains the characters of a string. Given this low-level function we have seen how several useful routines can be built that use it. Most relatively modern programming languages contain a built-in library containing a collection of such low-level functions which make it unnecessary for the programmer to deal directly with the arrays themselves. (Some of these are discussed below.) In very modern programming languages, especially object-oriented languages such as C++, Java and SmallTalk, the programmer can use 'string objects' that encapsulate both the underlying array implementation and all the useful routines that form the associated function library. The material presented here will not deal with objects but with library functions.

Standard string processing functions

A function library devoted to string processing would contain many functions. We will assume that the ones discussed below are available to us in our algorithm design work.

trimString

A function 'trimString' is used to remove all the spaces from the front and the rear of a string. It is useful in user-interface applications because research shows that when human users enter text into a computer program they often inadvertently type some spaces before they start the actual text and also at the end of the text.

The 'trimString' function does not remove spaces that are embedded inside a string; it removes only leading and trailing spaces. So if we have a string called 'myMessage' which holds the characters shown in Figure 4.23, then 'trimString' will remove the two spaces before the 'H' and the three spaces after the 'e', but it will not remove the space between the 'p' and the 'm'.

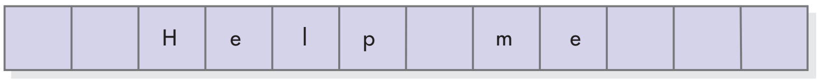


Figure 4.23

The 'trimString' function accepts a string as an argument, trims it, and returns the resulting string. Here is a sample algorithm that shows its use.

```
BEGIN trim experiment
    msg is an array of characters indexed from 1 to 100
    print 'Please enter a message'
    read msg from the user
    set msg to trimString with msg
    print msg
END
```

When this algorithm executes, the user enters a message that is trimmed of leading and trailing spaces and simply displayed.

compareStrings

A function called 'compareStrings' is useful to test the alphabetical order of two strings. This function accepts two strings and compares them to see which one would come first in an alphabetical listing. It returns a number that is 1, –1 or 0, indicating that the first string comes after the second, the second comes after the first, or that the two strings are identical. This is quite hard to understand at first so some examples are given below.

Suppose that we have two strings, "`Freda`" and "`Anne`".
To simulate the way the 'compareStrings' function works, we need to ask which one of these would be farthest down in an alphabetical listing. The answer would be 'Freda' because 'Anne' would be near the top of such a listing and 'Freda' would be lower down. If we use the compareStrings function like this:

```
set result to compareStrings with 'Freda' and 'Anne'
print result
```

then the printed result would be 1 (the first string is lower than the second in alphabetical order). However, if we were to write

```
set result to compareStrings with 'Anne' and 'Freda'
print result
```

the printed result would be –1 because the first string ('Anne') is higher in alphabetical order. Finally, if we were to write

```
set result to compareStrings with 'Freda' and 'Freda'
print result
```

the printed result would be 0 because the two strings are identical.

toLowerCase, toUpperCase

These two functions convert all the characters of any string into lower case or upper case. They return a string that contains the new version of the argument. They are very straightforward and can be used like this:

```
BEGIN case experiment
    set sampleWord to 'My homework was stolen'
    set upword to toUpperCase with sampleWord
    set loword to toLowerCase with sampleWord
    print sampleWord
    print upword
    print loword
END
```

The output from this algorithm would be:

```
My homework was stolen
MY HOMEWORK WAS STOLEN
my homework was stolen
```

stringContains

This is a useful function to have in a library. It takes two strings as arguments and checks if the first string contains the second. If the first string does contain the second, the function returns 'true'; otherwise it returns 'false'.

This function can be used effectively in some searching algorithms. The user enters a word or phrase and the algorithm then searches through a collection of text checking to see if the user's word appears.

As a preliminary example, imagine that we have a string containing the word 'rectangle'. If we call the 'stringContains' function with this string as its first argument and 'angle' as its second argument, the function will return 'true', because 'angle' is contained within 'rectangle'. The following sample algorithm shows how this would be done.

```
BEGIN contains experiment
    set foundIt to stringContains with 'rectangle' and 'angle'
    IF foundIt is true
    THEN
        print 'rectangle contains angle'
    ELSE
        print 'rectangle does not contain angle'
    ENDIF
END
```

An algorithm that uses the string library

This section looks at an algorithm that uses the assumed string library to do some useful work. It concerns a miniature thesaurus application.

In all the pseudocode shown below, whenever a string is being created the word 'string' has been used as a type; for example:

```
myName is a string
```

This is instead of the longer version used with arrays, such as:

```
myName is an array of characters indexed from 1 to 100
```

This shorthand notation helps to reinforce the idea that we can now leave the low-level details behind when we have a function library ready to use.

The thesaurus application

A thesaurus is a collection of words and their synonyms. To use a printed thesaurus, we look for a particular word in the index, go to the page indicated and read the word's synonyms. For example, if we look up the word 'computer', we might find several words that mean the same thing or have meanings that are related, such as 'machine', 'calculator', 'PC', 'mainframe', 'processor', and so on. A printed thesaurus is useful when completing crossword puzzles.

Often a word-processing package will have an electronic thesaurus built into it. An author writing something using a word processor might use a word that he or she feels is not quite right for the piece. The built-in thesaurus can be used to look for a more appropriate word.

Our application will be a simplified version of an electronic thesaurus. Suppose we have a file called 'synonyms' stored on our system in which each line contains a set of synonyms. The first few lines of the file might look like this:

```
computer machine PC mainframe processor
doctor physician specialist PhD quack academic
house abode shelter home address bungalow flat unit
```

Notice that each line is a collection of words and that every word on a particular line is a synonym for the other words on that line. Also, the number of words on each line is not fixed.

In our simplified thesaurus application we will allow the user to enter a word, then we will read through the file searching for that word. If we find the word, we will display all the synonyms. Here is the algorithm.

```
BEGIN thesaurus
    searchWord is a string                                  // 1
    upSearchWord is a string                                // 2
    aLine is a string                                       // 3
    upALine is a string                                     // 4
    theFile is a file                                       // 5
    open theFile as file 'synonyms'                         // 6
    print 'Please enter a word'                             // 7
    get searchWord from user                                // 8
    set upSearchWord to toUpperCase with searchWord         // 9
    REPEAT
       read aLine from theFile                             // 10
       set upALine to toUpperCase with aLine               // 11
       IF stringContains with upALine and upSearchWord     // 12
       THEN
          print aLine                                      // 13
       ENDIF
    UNTIL end-of-file theFile                              // 14
END
```

This thesaurus algorithm is reasonably modest in size. This is because much of the hard work is actually done by the string library functions used on lines 9, 11 and 12.

The algorithm starts by creating several string variables. On line 1 the variable 'searchWord' is created. It is used to hold the word the user enters. On line 2 the variable 'upSearchWord' is created and is used to hold an upper-case version of it. Similar work is done for strings that hold the data read from the file, on lines 3 and 4.

The preliminary work of this algorithm is to open the file ready to start reading data and to fetch the user's word (lines 6, 7 and 8). Let us assume that the user has entered the word

```
Specialist
```

On line 9 we call the standard string library function 'toUpperCase' which converts this word to all upper-case characters. The result is stored in the variable 'upSearchWord', which would contain

```
SPECIALIST
```

Following this (line 10 in the algorithm), we read the next string of words from the file. This is an entire line of the file containing one complete set of synonyms. The first time this is done, the variable 'aLine' would contain

```
computer machine PC mainframe processor
```

Next we use the 'toUpperCase' function again so that the variable 'upALine' contains

```
COMPUTER MACHINE PC MAINFRAME PROCESSOR
```

The next line, line 12, is really the heart of the algorithm. It calls the standard string function 'stringContains' and passes the data from the file and the user's word to it. You should recall that this function checks to see if the second string is contained in the first, so in this case it checks to see if

```
SPECIALIST
```

is contained in

```
COMPUTER MACHINE PC MAINFRAME PROCESSOR
```

Of course 'SPECIALIST' is not found, and so the 'stringContains' function returns 'false'. This means that the print statement on line 13 is not executed.

The REPEAT-UNTIL loop condition is also 'false' at this stage because there is still more data in the file to be read. So the lines 10 to 14 in the algorithm are repeated. When the second set of synonyms in the file is read, the 'stringContains' function call on line 12 will return 'true' (you should verify this for yourself), and the algorithm will print the words

```
doctor physician specialist PhD quack academic
```

which are the synonyms the user is looking for.

Exercise 4.6

1 Why is a library of string functions preferable to writing the low-level functions from scratch?

2 What is the underlying data structure that supports string type variables?

3 The complete algorithm for the low-level 'concatenate' function was given in this section. Write your own version of a similar low-level function called 'indexOf' which accepts two

arguments. One argument is a string and the other is a single character. The function should return the index of the first cell in the string which contains the given character. If the character is not found in the string, the function should return −1.

4 A simple function that is useful in programs that deal with strings is one called 'stringLength'. It simply returns the number of characters stored in a string. Write this function taking care to note that the number of characters in a string is not always equal to the number of cells in the underlying array.

5 Write an algorithm in which the user is allowed to enter two messages (strings). The algorithm should then print the two strings in uppercase letters.

6 Write an algorithm (or extend the one from 5) that allows the user to enter two strings. The algorithm should then print both strings, with the longest string appearing first in the output.

7 Write an algorithm in which the user enters two messages and they are printed in alphabetical order. Use the 'compareString' library function in your solution.

8 Write an algorithm in which the user enters two strings. The algorithm should determine whether the second string is part of the first, and output a suitable message. Use the 'stringContains' library function in your solution.

9 Does your solution to exercise 8 work correctly if the first entered string contains all lower-case characters and the second all upper-case letters? If it does not, modify your solution so that the case of the characters in the strings does not affect the outcome.

10 Write a bubble sort routine that sorts an array of strings. You should attempt to write the routine from scratch, and if you cannot complete it, refer to the bubble sort given in the previous section. The bubble sort given earlier compares numerical cell values with the operator '<' (less than). The string version will have to use the 'stringCompare' library function instead.

Records and collections

This section introduces records, which are units of information that can be stored, searched and retrieved by computer software. The work on records takes us closer to modern computing in at least two ways: it involves concepts that are similar to those found in database management systems, and it involves concepts that are found in object-oriented programming.

We will first look at the fundamental structure of records and an algorithm syntax for defining them. They will then be combined with what we already know about arrays to form applications that hold and manage collections of records—'containers'.

Records

In the algorithm design seen so far, each simple variable we declare can hold only one value and that value can be of only one type. If we wanted to store a person's age, we could declare a variable like this:

```
age is a whole number
```

We could assign a value to the age variable, like this:

```
set age to 17
```

It is obvious that with this variable we can store only one person's age and the age must be a whole number. If we want to store a person's name, which is a string, we could declare and assign another variable:

```
name is a string
set name to 'Jim Morrison'
```

The variable 'age' and the variable 'name' are two completely separate variables. A computer program would maintain absolutely no connection or relationship between them.

However, human beings might see things differently. We might see the age and the name as connected because they refer to two pieces of information about the same person. To allow a relationship between separate variables in computer algorithms, we can declare a ‘record’ that contains both variables, like this:

```
Person is a record containing
    name, a string
    age, a whole number
END
```

A person’s age and name have been gathered together and declared inside a record. Programmers often say that the record ‘encapsulates’ the two variables, because the Person record is like a capsule. The separate variables inside the record are called ‘fields’ or ‘members’ or ‘attributes’. So, a reasonable definition of a record is that it is the encapsulation of several variables as fields in the one structure.

Database terminology

The terms ‘record’ and ‘field’ used above are taken directly from database terminology. A database management system would maintain ‘tables’; each table would consist of ‘records’, and each record would contain several fields.

A library that wanted to use a database to hold information about its books would first decide on what data to hold about each book. Each piece of data would be a field, such as ‘title’, ‘author’, ‘number of pages’ and so on. Each book’s data would be encapsulated in a database ‘record’. To do the same kind of work in our algorithm description language, we would write:

```
Book is a record containing
    title, a string
    author, a string
    numPages, a whole number
END
```

A collection of database records is usually called a ‘table’. The algorithms in this book will be restricted to using an array to hold a collection of records.

Object-oriented programming (OOP) terminology

Similar concepts are used in OOP. The various separate pieces of data are collected into an ‘object’ and are called ‘members’ or ‘attributes’. In OOP, functions as well as data can be encapsulated inside objects, so each object is a collection of some data, representing information, and some functions, representing certain behaviour. Whereas we use the word ‘record’ in our algorithm description syntax, OOP uses the term ‘object’. In OOP a ‘class’ is used as a template or definition of what data and behaviour a particular type of object can hold.

The term used to describe a collection of objects in OOP is a ‘container’. A container class defines a host object that holds as its data an array of some other kind of objects and has encapsulated in it functions that allow the collection to be managed.

Using records

There is a simple way of specifying a record; for example:

```
Date is a record containing
    day, a whole number
    month, a whole number
    year, a whole number
END
```

It is important to realise that this is a definition of what all Date records will look like. It is not the declaration of any particular date. In this sense, when we define a record we are doing what OOP does with a class—defining a ‘template’ that simply shows how a particular type of record is structured.

To create a particular Date record to hold some specific values, we can write:

```
anzac is a Date record
```

which means that the single word 'anzac' is a variable holding one complete Date record. We can access the fields (or attributes) of this record where the three attributes are set to suitable values:

```
set anzac.day to 25
set anzac.month to 4
set anzac.year to 2001
```

Note that a single dot is placed between the name of the record variable (anzac) and the name of one of its fields. This kind of dot syntax has been used in languages such as C and C++ for many years. The syntax is also used in many more recent languages such as Java, JavaScript and XML Schema.

Nested records

Records are a neat way of holding a set of values together. The Date record holds together all three values for a given date. The Person record defined earlier held together two values for a particular person—the person's name and age.

It makes sense when defining a Person record to include the person's date of birth instead of their age, because any age value would need changing each year; a date of birth could remain constant. The Person record could encapsulate a name (another simple variable) and a date (itself a record type). We say that the Date record is 'nested inside' the Person record, or more simply that the Person record has a Date record as one of its fields. The definitions of the Date and Person records are as follows:

```
Date is a record containing
    day, a whole number
    month, a whole number
    year, a whole number
END
Person is a record containing
    name, a string
    dob, a Date
END
```

A Person variable can be declared as:

```
tom is a Person record
```

Values for some of the attributes can be set as:

```
set tom.name to 'Tom Tank'
```

This person's date of birth can be accessed as:

```
tom.dob
```

So, to set values for the day, month and year for this date of birth, we could write:

```
set tom.dob.day to 12
set tom.dob.month to 6
set tom.dob.year to 1988
```

Examples using single records

To see how individual records can be used in algorithms, consider a record type that holds the name of a company, the current price of its shares and the number of shares a person owns in that company.

```
Stock is a record containing
    coName, a string
    sharePrice, a real number
    numShares, a whole number
END
```

We can create one of these records for 100 shares in the company 'ABC Bank', with a share price of $18.50.

```
stock1 is a Stock record
set stock1.coName to 'ABC Bank'
set stock1.sharePrice to 18.50
set stock1.numShares to 100
```

For a company called 'Pata Mining' the record might look like this:

```
stock2 is a Stock record
set stock2.coName to 'Pata Mining'
set stock2.sharePrice to 23.50
set stock2.numShares to 1200
```

A function that calculates the value of some shares for a company can be written to accept a Stock record and a quantity and return a dollar value.

```
BEGIN stockValue with someStock
    value is a real number
    set value to someStock.price × someStock.numShares
    return value
END
```

A function could also be written to calculate the brokerage when some shares are traded. The term 'brokerage' refers to a fee that is charged whenever shares are bought or sold. In our example, the brokerage fee will be 0.3% of the value of the shares traded (for values over $10 000) or a flat fee of $30 for values less than $10 000. Note that this function calls the previous one to find the value of the trade.

```
BEGIN getBrokerage with someStock
    fee is a real number
    value is a real number
    set value to stockValue with someStock
    IF value < 10000
    THEN
        set fee to 30
    ELSE
        set fee to value × 0.003
    ENDIF
    return fee
END
```

These two functions can be used together in an algorithm that estimates the total cost of buying some shares. This algorithm requests that the user enter the company name and the price of the shares and also how many shares are required. It then displays the value of the shares, the brokerage fee and the total amount to be paid.

```
BEGIN share trading algorithm
    print 'What is the company name?'
    get name from the user
    print 'What is the current price?'
    get price from the user
    print 'How many shares do you want to buy?'
    get size from the user
    Trade is a Stock record
    set Trade.coName to name
    set Trade.sharePrice to price
    set Trade.numShares to size
    set value to stockValue with Trade
    set fee to getBrokerage with Trade
    set total to value + fee
    print 'Estimate of costs when buying' Trade.coName
    print 'The shares will cost ' value
    print 'The brokerage will be ' fee
    print 'The total to be paid is ' total
END
```

If this algorithm was a computer program and we were to execute it, the results would look something like this. (The bold text indicates the user's input.)

```
What is the company name? Norick Engineering
What is the current price? 15.60
How many shares do you want to buy? 800
Estimate of costs when buying Norick Engineering
The shares will cost 12480.00
The brokerage will be 37.44
The total to be paid is 12517.44
```

Collections

A 'collection' is a general term that refers to the storage and management of several or many records. As stated earlier, a database management system holds a collection of records in a table and provides several standard routines for managing the records (such as adding new records, searching and retrieving records, and changing the values stored in records). In object-oriented programming, specific objects defined in 'container classes' do the same kind of work.

In this section collection storage is simulated by creating an array of records, and the management of the collection is simulated with several functions. The example used is a collection of Stock records which can be thought of as a 'share portfolio'. The example is relatively simple and could be used as part of a personal share investment package.

Defining the storage

To create an array of 20 Stock records in our algorithms, we write:

```
portfolio is an array of Stock records indexed from 1 to 20
```

We will start with no Stock records in the collection and allow users to add new Stock records one at a time, so we will need to keep track of how many records are stored at any time. We can do this with a simple variable:

```
numRecords is a whole number
```

Any application must start with an empty collection, so this variable will have to be initialised to zero:

```
set numRecords to 0
```

Defining the functions

To add a new record to the collection, we need to know the name of the company, the number of shares held and the price of each share. The following function takes this data as arguments, creates a new Stock record with it and inserts the record into the next available cell in the array. The next available cell can be determined by the value of the 'numRecords' variable:

```
BEGIN addStock with name and size and price
    IF numRecords = length of portfolio
        return false
    ENDIF
    someStock is a Stock record
    set someStock.coName to name
    set someStock.numShares to size
    set someStock.sharePrice to price
    set portfolio[numRecords+1] to someStock
    increment numRecords
    return true
END
```

You should verify that this function works correctly. The first time it is called, the variable 'numRecords' will be zero. Can you see that the new Stock record will be inserted into cell 1 of the array and that the value of 'numRecords' will be incremented to 1? Also notice that this function returns either 'true' or 'false' (it's a boolean function). It will return 'true' only if the new Stock record has been successfully added into the array. It will return 'false' if the array is already full and the new record cannot be added.

A function that searches for a given record will accept a company name as an argument. It will then loop through the array, comparing this name with the company name of each stored record. When a match is found, the appropriate record will be returned to the function's caller. If no match can be found, a dummy record will be returned.

```
BEGIN findRecord with name
    set counter to 1
    WHILE counter < length of portfolio
        IF portfolio[counter].coName = name      // (1)
        THEN
            return portfolio[counter]      // (2)
        ENDIF
        increment counter      // (3)
    ENDWHILE
    dummy is a Stock record
    set dummy.coName to 'No Name'
    set dummy.sharePrice to 0
    set dummy.numShares to 0
    return dummy
END
```

To test this function, assume that there are exactly three records stored in the array: Telstra, Coles and Qantas. The function is called like this:

```
set someStock to findStock with 'Coles'
```

What Stock record will be returned from the function and captured into 'someStock'?

The function will start with its counter being 1. It will then compare the name 'Coles' with the company name of the first record ('Telstra'). Of course, these names do not match and so the function increments the counter (at line 3) and loops again. The second time through the loop, it compares the name 'Coles' with the company name in the second record ('Coles') and does find a match. So the whole Stock record in the second cell of the array will be returned.

There are two basic changes that an investor might want to make to a parcel of shares: to buy some more of them or to sell some of them. The following two functions provide these services for any individual Stock record. In both cases the functions accept a Stock record and a number of shares.

```
BEGIN buyShares with someStock and quantity
    IF quantity > 0
    THEN
        increment someStock.numShares by quantity
    ENDIF
END
BEGIN sellShares with someStock and quantity
    IF quantity > 0 AND quantity < someStock.numShares
    THEN
        decrease someStock.numShares by quantity
    ENDIF
END
```

Share investment algorithm

The small but complete algorithm presented below interacts with the user to provide a share portfolio management application.

Note that the array and the number of records variable are created first and are outside of all modules. This means that they are directly available to all modules; they have global scope in our algorithm.

The main module appears first. Its role is to present a menu to the user, to fetch the user's menu choice and to direct the work out to other, smaller modules that do more specific tasks.

```
portfolio is an array of Stock records indexed from 1 to 20
numRecords is a whole number
set numRecords to 0
BEGIN Share Investment Application
    option is a whole number
    REPEAT
        print '1. Add a new stock'
        print '2. Find a stock'
        print '3. Sell some shares'
        print '4. Buy some extra shares'
        print '5. Show total portfolio value'
        print '6. Quit'
        get option from the user
        CASEWHERE option is
            1: addNewStock
            2: findAndShow
            3: findAndSell
            4: findAndBuy
            5: showTotal
            6: print 'Application terminated'
            OTHER: print 'An illegal menu option was entered'
        ENDCASEWHERE
    UNTIL option = 6
END
```

Adding a new stock is done in the 'addNewStock' sub-module (function), like this:

```
BEGIN addNewStock
    print 'Enter the company name'
    get name from the user
    print 'Enter the share price'
    get price from the user
    print 'How many shares?'
    get size from the user
    set itWorked to addStock with name and size and price
    IF itWorked = true
    THEN
        print 'Stock successfully added'
    ELSE
        print 'Can't add stock, portfolio is full'
END
```

We already have a 'findStock' function, shown earlier. This 'findAndShow' module provides some user interaction and calls the 'findStock' function as necessary.

```
BEGIN findAndShow
    print 'Which stock name?'
    get name from the user
    set someStock to findStock with name
    IF someStock.coName = 'No Name'
    THEN
        print 'Sorry, can't find a match'
    ELSE
        print 'Company name is ' someStock.coName
        print 'The price is ' someStock.sharePrice
        print 'You have ' someStock.numShares 'shares'
    ENDIF
END
```

In order to sell shares, we must first find the correct record, so our 'findAndSell' module also calls 'findStock'. It then calls 'sellShares', developed earlier, to process the selling transaction.

```
BEGIN findAndSell
    print 'Which stock name?'
    get name from the user
    set someStock to findStock with name
    IF someStock.coName = 'No Name'
    THEN
        print 'Sorry, can't find a match'
    ELSE
        print 'How many shares to sell?'
        get amount from the user
        IF sellShares with amount = true
        THEN
            print 'Transaction confirmed'
        ELSE
            print 'Can't sell that many shares'
        ENDIF
    ENDIF
END
```

A module for buying shares is almost identical.

```
BEGIN findAndBuy
    print 'Which stock name?'
    get name from the user
    set someStock to findStock with name
    IF someStock.coName = 'No Name'
    THEN
        print 'Sorry, can't find a match'
    ELSE
        print 'How many shares to buy?'
        get amount from the user
        IF buyShares with amount = true
        THEN
            print 'Transaction confirmed'
        ELSE
            print 'Can't buy that many shares'
        ENDIF
    ENDIF
END
```

Finally, to complete the algorithm, there is a module that prints out the total value of all the shares in the collection. It loops over all the stock records in the array and calls the 'stockValue' function on each one to retrieve its value. This is accumulated into a total.

```
BEGIN showTotal
    set total to 0
    FOR counter goes from 1 to numRecords
        increment total by stockValue with portfolio[counter]
    ENDFOR
    print 'Your portfolio is worth ' total
END
```

Exercise 4.7

1 How does the word 'encapsulate' relate to a record data type?

2 In database terminology a record contains 'fields'. What is the equivalent terminology in object-oriented programming for the data values stored in an object?

3 In object-oriented programming an object (or record) can hold other things as well as data. What are these other things?

Exercises 4, 5 and 6 refer to the following record definition:

```
BankAccount is a record containing
    ownersName, a string
    balance, a real number
END
```

4 Write the declaration of two 'BankAccount' records, one for a savings account and one for an investment account.

5 Set the name of the owner of the savings bank account to 'Wally Wallpaper' and the owner's name of the investment account to 'Bill Gates'.

6 Write an algorithm fragment that compares the balance of one account with the other and prints a message containing the name of the owner of the account that contains the highest balance.

7 What term is used to describe the situation when one of the fields of a record is itself a record type?

Exercises 8, 9 and 10 refer to the following record declarations:

```
Date is a record containing
    day, a whole number
    month, a whole number
    year, a whole number
END
    DiaryEvent is a record containing
    theDate, a Date record
    message, a string
END
```

8 Declare a variable of the record type 'DiaryEvent'.

9 Into the record variable declared in question 8, store suitable values to stand for the event 'Sydney Olympics' which occurred on the date 15/9/2000.

10 Write an algorithm fragment that adds 10 days to the 'Date' variable in the 'DiaryEvent' variable.

11 Write a function that accepts a DiaryEvent record as an argument. The function is to test the month stored in the date record within the DiaryEvent and print the DiaryEvent message if the month is after June (that is, value 6). If the month is not after June, the function is to print 'This event is out of date'.

12 Declare an array of 10 DiaryEvents and store an event with the date 1/1/2001, and the message 'First day of the new century', into the first cell of the array.

13 Using the portfolio application as a guide, create several functions that manage an array of DiaryEvents.

14 Write an algorithm that allows the user to interact with the array of DiaryEvents in the same way as was done with the portfolio algorithm. In your algorithm the user will be interacting with a computerised diary, adding events, finding events, listing events and so on.

Team Activity

Your team has been contracted to add some functionality to an existing software product ready for implementation in a new version for the next release. The existing product is similar to a word-processing package, but instead of saving its files in a proprietary format it outputs files with HTML markup. These files are intended to become webpages. The product already contains basic word-processing features such as text entry and formatting styles. But it lacks some other features that are useful to authors, such as sorting a table of contents, prompting with synonyms from a thesaurus and auto-text lookup.

The existing product interface allows an author to highlight any word in a document and then to choose a command from a menu. Your team is asked to write several separate modules each of which accepts the highlighted word as input and performs some useful functions with it. The required modules are listed below.

1 A module to convert the case of the author's highlighted word. Given a word as input, design and write the algorithm for a module that outputs the word in all uppercase letters. Hint: Use functions from the assumed string library in your solution.

2 A module to allow the author to add any highlighted word to a collection of often-used words. Given a word as input, design and write the algorithm for a module that appends it onto any array of words for later use. You need to allow for a maximum of 100 words in total for this collection. Hint: Use the principles discussed in the section of this chapter on collections.

3 A module to allow an author to get a list of synonyms for any highlighted word. Given a word as input, design and write an algorithm for a module that locates the word in a large collection of words and outputs all its synonyms. Hint: Create a record structure that contains several words that are synonyms for each other and store several of these records in a collection. When the input word is known, search the collection for the first record containing the word and output all the other words in the current record.

As a team activity, you should approach the above with due discussion and thought. Plan to have all your modules implemented as reusable functions in the style shown in this chapter.

Review exercises

1 Write a description in your own words of the fundamental principles involved in reusable functions. Include in your description some indication of how functions are called, how they get their input data, where their output goes and what happens when one function is called and then it calls another function.

2 Collect the names of the students in your class or group. Write these names, in the order they are collected, onto a large white-board. With an eraser and a white-board marker rearrange the names so that they are in alphabetical order. Use whatever strategy you like to do this and involve the whole group in suggestions about which names should be erased and rewritten in different positions.

3 Do the activity in exercise 2 again, this time following the logic of a bubble sort.

4 Do the activity in exercise 2 again using selection sort logic.

5 Write a function that accepts four whole numbers as arguments and returns the number that is smallest.

6 Convert your solution to exercise 5 to a function that accepts four strings and returns the one that is shortest.

7 Write a function that accepts an array of whole numbers and returns the difference between the biggest number and the smallest.

8 In a diving competition there are ten judges who each scores a particular dive with a number from 0 to 10. To avoid bias, the smallest score and the largest score are ignored and the diver's final score is the average of the remaining eight scores. Write an algorithm to perform this work. Make sure you employ separate functions to do any separate parts of the work so your algorithm has a modular structure. You can use the array traversal work from this chapter as a guide.

9 In a game similar to lotto, a player chooses six unique numbers in the range 1...44. Another six unique numbers in the same range are chosen by drawing numbered marbles from a barrel—these become the six winning numbers. Write an algorithm that calculates how many of the players' numbers match the winning numbers. You can use an array to hold each group of six numbers and write a function that accepts the two arrays and returns the number of matches.

10 Design and implement an algorithm to hold and manage a collection of music CDs. The relevant information on each CD can be held in a record with a field for the title, a field for the artist, a field for the type of music and so on. Your algorithm should allow information about a new CD to be added to the collection, the information on any CD to be located and displayed given an artist's name, and a list of all CDs matching a given style of music to be produced. If you wish, you can model your algorithm on the portfolio algorithm given in the last section of this chapter.

11 An insertion sort is a third method of sorting. It divides the array to be sorted into two parts: the unsorted part and the sorted part. At the start, the first element of the array is assumed to be the sorted list. The second element is then put into its place in the sorted list that then consists of the first two elements in order. The third element is then put in its place in the sorted list. In this way the sorted list grows and the unsorted part shrinks.

Using some of the templates presented in this chapter, write an algorithm to perform an insertion sort.

Chapter summary

- Programming and algorithm design can only be learnt by a significant amount of practice. Reading about programming, watching people do it or listening to people talk about it is not enough.
- Accomplished programmers keep a stock of design patterns. A design pattern is a reuseable fragment of logic.
- Structured algorithms are based on a single top-level controlling module which farms out various activities to small, separate module or functions.
- Structured algorithms can be devised by planning the main controlling module first and then creating the sub-modules (top-down design) or by defining the sub-modules first and then writing a controlling main module later (bottom-up design).
- Data can be held in variables that are available to all modules (global data) or in variables that are restricted in use to separate modules (local variables).
- Global data has some efficiency benefits in algorithms, but sometimes it is difficult to debug programs in which global data is changed. This is because it is not always easy to find which module changed the global data incorrectly.
- Local variables are generally considered a safer way to hold data.
- Algorithms should be thoroughly tested using carefully chosen sample data. Desk checks and trace tables are useful for testing algorithms.
- Selection control structures should be tested with sample data that makes their true branch fire, data that makes their false branch fire and data that meets the boundary condition.
- Testing can be carried out by the programmer and also by users in the field.
- All algorithms should contain a reasonable level of internal documentation (comments) which help other programmers who might have to modify the code.
- Software products require external documentation for users.
- Assumptions should be listed as part of internal documentation, especially when a problem description is ambiguous.
- Functions can define their own local variables and should always accept their input data via arguments (or parameters).
- Functions should deliver their output by returning it to whichever code calls on them.
- When a function is called, the calling code is suspended until the called function returns.
- Functions always return control to the code that calls them.
- A function should attempt to perform just one well-defined process.
- A function should not share data with other functions, except for data that is explicitly passed to it via arguments.
- When a function's 'return' statement is executed, that function terminates immediately.
- An array is a data structure that contains a sequence of values and all the values must be of the same type.
- It is impossible to extend the length of an array once it has been created. It is not necessary to fill all cells in an array with values.
- Many useful activities with arrays involve traversing the array, visiting every cell from the first to the last. A FOR loop structure is most appropriate for traversing arrays.
- A linear search of an array involves a simple traversal and a test of each cell's value to see if it matches the value being searched for.

Chapter summary

- A binary search is much more efficient than a linear search but it can only be done if the values in the array are in sorted order.
- A binary search involves repeatedly checking the middle value of a range of cells and adjusting the range after each check is done.
- A bubble sort involves swapping adjacent pairs of cell values in an array until the largest (or smallest) value 'bubbles' to one end of the array.
- A selection sort starts by finding the smallest value (or the largest) in an array and swapping it with the first cell.
- Strings are actually implemented as arrays of characters.
- Modern languages usually allow the programmer to access a standard collection of functions that do many of the useful string processing routines.
- A record is a structure in which several variables, each of different types, can be held together.
- An array of records is a suitable data structure for holding a collection of information similar to a database.

chapter 5
Implementation of software solutions

Outcomes

A student:

- explains the interrelationship between hardware and software (H 1.1)
- differentiates between various methods used to construct software solutions (H 1.2)
- describes how the major components of a computer system store and manipulate data (H 1.3)
- explains the relationship between emerging technologies and software development (H 2.2)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- constructs software solutions that address legal, social and ethical issues (H 3.2)
- applies appropriate development methods to solve software problems (H 4.2)
- applies a modular approach to implement well-structured software solutions and evaluates their effectiveness (H 4.3)
- applies project management techniques to maximise the productivity of the software development (H 5.1)
- creates and justifies the need for the various types of documentation required for a software solution (H 5.2)
- selects and applies appropriate software to facilitate the design and development of software solutions (H 5.3)
- communicates the processes involved in a software solution to an inexperienced user (H 6.2)
- uses a collaborative approach during the software development cycle (H 6.3)

Students learn about:

Interface design in software solutions

- the design of individual screens, including:
  - identification of data required
  - current popular approaches
  - design of help screens
  - audience identification
  - consistency in approach

Language syntax required for software solutions

- use of BNF, EBNF and railroad diagrams to describe the syntax of new statements in the chosen language
- commands incorporating the definition and use of:
  - multi-dimensional arrays
  - arrays of records
  - files (sequential and relative/random)
  - random number generators

The role of the CPU in the operation of software

- machine code and CPU operation
  - instruction format
  - use of registers and accumulators
  - use of program counter and fetch–execute cycle
  - addresses of called routines
  - linking, including use of DLLs

Translation methods in software solutions

- different methods include:
  - compilation
  - incremental compilation
  - interpretation
- the translation process
- advantages and disadvantages of each method

Program development techniques in software solutions

- structured approach to a complex solution, including:
  - one logical task per subroutine
  - stubs
  - flags
  - isolation of errors
  - debugging output statements
  - elegance of solution
  - writing for subsequent maintenance
- the process of detecting and correcting errors, including:
  - syntax errors
  - logic errors
  - peer checking
  - desk checking
  - use of expected output
  - run time errors, including:
    arithmetic overflow, division by zero and accessing inappropriate memory locations
- the use of software debugging tools, including:
  - use of breakpoints
  - resetting variable contents
  - program traces
  - single line stepping

Documentation of a software solution

- forms of documentation, including:
  - process diary
  - user documentation
  - self-documentation of the code
  - technical documentation including source code, algorithms, data dictionary and systems documentation
  - documentation for subsequent maintenance of the code
- use of application software to assist in the documentation process
  - use of CASE tools

Hardware environment to enable implementation of the software solution

- hardware requirements
  - minimum configuration
  - possible additional hardware
  - appropriate drivers or extensions

Emerging technologies

- hardware
- software
- their effect on:
  - human environment
  - development process

Students learn to:

- select either a sequential or an event-driven approach and an appropriate language to effectively solve the problem
- design and evaluate effective screens for software solutions
- utilise the correct syntax for new commands using the metalanguage specification
- produce syntactically correct statements
- implement a solution utilising a complex algorithm
- recognise and interpret machine code instructions
- choose the most appropriate translation method for a given situation
- utilise the features of both a compiler and an interpreter in the implementation of a software solution
- justify the use of a clear modular structure with separate routines to ease the design and debugging process
- use drivers to test specific modules, before the rest of the code is developed
- differentiate between the different types of errors encountered during the testing phase
- recognise the cause of a specific error and determine how to correct it
- effectively use a variety of appropriate error correction techniques to locate the cause of a logic error and then correct it
- produce user documentation (utilising screen dumps) that includes:
  - a user manual (topics presented in order of difficulty)
  - a reference manual (all commands in alphabetic order)
  - an installation guide
  - a tutorial to introduce new users to the software
- identify the personnel who would be likely to use the different types of documentation
- recognise the need for additional hardware
- assess the effect of an emerging technology on society

Interface design in software solutions

A software solution can only be as good as the interface that it presents to the user. There is no point in spending a great deal of time and effort in developing a computer solution to a problem if the user does not want to use it or cannot use it.

There are a number of factors that contribute to an effective user interface and we will investigate some of these in the next sections. Some of the factors are those that relate to the way in which the user interacts with the program, for example user screens. Other factors may be determined by the types of data that have to be processed; for example, the processing of sound will mean that the program may need to be able to input from microphones or output to speakers.

The design of the human interface will also need to take into consideration human factors such as ergonomics and equity. The program's target audience can also have an effect on the interface. For example, an audience of preschool children will not be able to use a highly text-based interface since their reading skills are not developed sufficiently for them to use it.

The design of individual screens

The screen is still the most common method of communication between the user and a program. It is therefore essential that the design of the screen be given a high priority in the development process.

The Preliminary Course examined the elements of good screen design. Here those elements will be put to work.

A number of factors will determine the overall design. These factors include the type of data to be presented, the type of audience the program is aimed at, assistance required by the user and consistency between screens.

Identification of data required

Screens are designed to present data, either input data or output data. It is important for the developer to have a clear understanding of how data items need to be presented to the user. The context in which data is displayed, if carefully chosen, can help the user relate to the process being carried out.

For example, as part of a library program a screen is required to display the results of a search. The data elements required by this screen will include the search criteria, a list of the matches to the search item and a means of gaining more detail about each of the matches to the search.

The type of data that is input will also have an effect on the programming model chosen. A sequential programming environment is usually chosen when the data items to be processed come from sources outside the system. Event-driven languages are generally used when the data to be processed is generated within the system, for example by sensors.

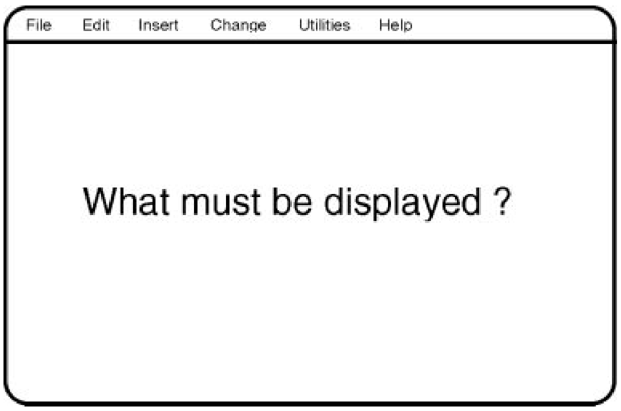


Figure 5.1 The identification of data items that form the basis for a screen is an important factor in the success of the screen and the whole program.

Current popular approaches

A number of tools are currently available that assist with the process of screen design. These tools range from simple graphics programs that can be used solely for the design process through to the use of programming languages, such as Visual Basic, that provide an integrated system of designing and implementing the design of the screen.

Other development environments, especially database management systems, also provide an integrated report and screen design system.

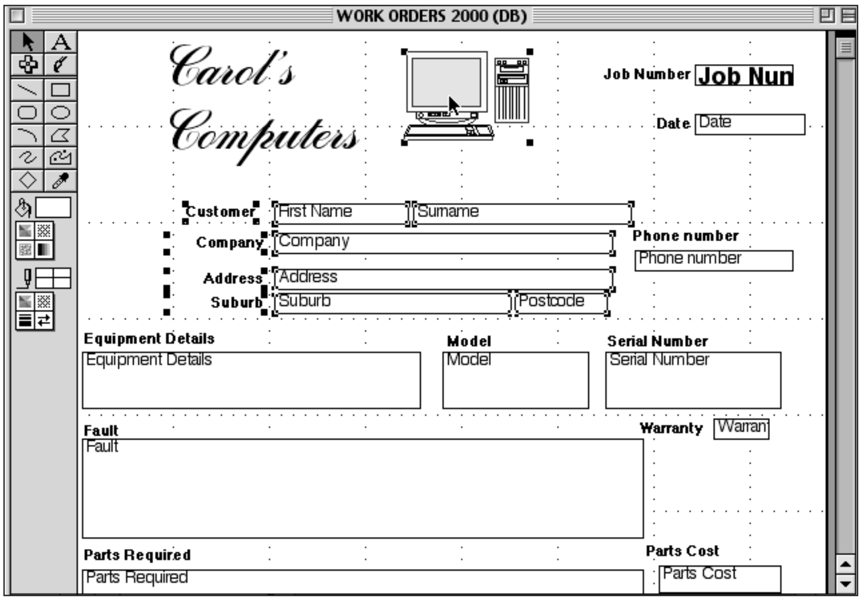
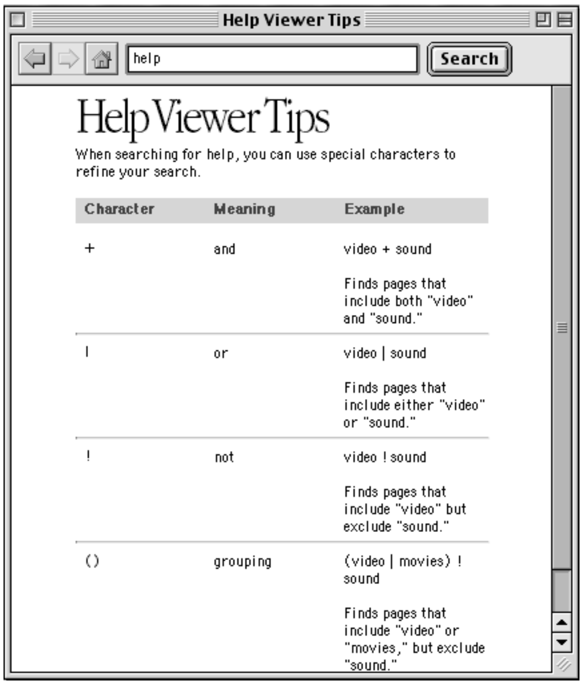


Figure 5.2 Database management systems, such as those found in integrated packages, provide a screen and report design environment.

Design of help screens

Design of help screens plays an important part in the overall process of software development. The help system should be designed to encourage the user to seek assistance immediately a problem is encountered.

Help screens may need to be designed to occupy a whole window, or they may occupy only a part of the window. A number of factors will determine which type is appropriate. A full screen will not, for example, be appropriate if a short explanation of the purpose of a button is to be presented. However, if a detailed explanation of a process is to be presented, a full screen may be more appropriate. In some cases it is helpful if the work window is available by the help window, as the user can refer to the help to identify the various screen elements needed to solve the problem.



Help Viewer Tips

When searching for help, you can use special characters to refine your search.

Character	Meaning	Example
+	and	video + sound Finds pages that include both "video" and "sound."
|	or	video | sound Finds pages that include either "video" or "sound."
!	not	video ! sound Finds pages that include "video" but exclude "sound."
()	grouping	(video | movies) ! sound Finds pages that include "video" or "movies," but exclude "sound."

Figure 5.3 Help screens do not have to occupy the whole of a window in order to be useful.

Audience identification

Each screen in a program will have a target audience. If the screen is to be effective, the needs of that audience must be met. These needs will cover areas such as the organisation of the screen elements, the manner in which these elements are presented, the way in which the user interacts with the interface and the way in which help is provided.

The screen designer has to look at the audience needs in terms of the tasks that will be performed, the capabilities of the user, the needs of the system and any limitations that are imposed by the technology to be used. Some of the information about these factors can be gained by examining the systems' descriptions. Further information will come from observation, surveys and interviews with members of the expected audience. Additional information may also be gleaned from prototypes of the proposed interface.

Consistency in approach

As seen in the Preliminary Course, there must be consistency between screens within an application. Consistency is important as it allows the user to anticipate actions and placement. For example, if the printer menu is always placed in the top right of the screen, the user will automatically move to that spot when one of the printer functions is required.

The design rules should be created before the development process is undertaken. If the rules are followed when the screens are designed, they will be consistent. This is especially important when a team of programmers is employed.

Exercise 5.1

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

The main link between a __________ and the user is the user interface. This means that the user __________ has to be designed carefully if the program is to be __________. Some of the factors that contribute to an effective user interface are the __________ screens, type of __________ that are processed, the target __________, assistance for the __________ and __________ between screens.

2 Explain the special features that an interface would need for a text reading program for a visually impaired student.

3 Choose an application and examine its interface. Describe the target audience for the application. Comment on the suitability of this interface for the target audience. Describe the ways in which this interface could be improved.

4 Use a graphics program to design a drawing screen for a children's program that is aimed at preschool children. Describe any peripheral devices you would use with this interface. Give reasons for your choices in terms of the target audience.

5 A program is being designed for an interactive gift selection centre in a shop. This selection centre will use a touch screen to allow customers to choose from a range of items selected for various occasions. There will be a main navigation screen with other screens for birthdays, engagements, weddings, Mother's Day, Father's Day, Christmas, Easter and baby gifts. Write a set of design rules to be followed by the screen design team.

6 Create one of the screens from question 5, using your screen design rules.

Language syntax for software solutions

In order for a programmer to correctly code an algorithm in a particular language, the rules of that language must be followed. The most concise way of providing this information is by means of metalanguages. These are languages that describe the structure of other languages. The most common of the metalanguages in use for the description of programming languages are the syntax structure diagrams, BNF and EBNF. These descriptions also serve to provide the rules that the translator uses when converting the program from source code (the high-level language you use) to object code (the code that can be understood by the processor). You will learn more about this process later in the chapter.

Use of BNF, EBNF and railroad diagrams to describe the syntax of new statements in the chosen language

In the Preliminary Course you learned how the syntax (rules) of a language can be described by means of a metalanguage. The metalanguages you are required to use are BNF (Backus Naur Form), EBNF (Extended Backus Naur Form) and syntax structure diagrams (railroad charts). Before starting this section of work, we will revise the use of these methods.

Syntax structure diagrams are a graphical method of showing the structure of a language. Rectangles and circles are used to represent the elements of the language and lines join the elements. Rectangles are used to represent non-terminal symbols, that is, the language elements that are defined elsewhere in the description. Circles or rounded rectangles enclose terminal symbols. Terminal symbols are those language elements that appear as written.

Symbol	Meaning
○	Terminal symbols—these symbols are part of the language.
⬭	Terminal symbols—these symbols are part of the language.
▭	Non-terminal symbols—these symbols represent a language element defined elsewhere.

Figure 5.4 Three symbols are used to represent language elements in syntax structure diagrams.

In the Figure 5.5, which represents the syntax of an assignment statement, a terminal symbol is the pair of characters ':='; the elements 'identifier' and 'expression' represent non-terminal symbols as they would have been defined elsewhere in the description.

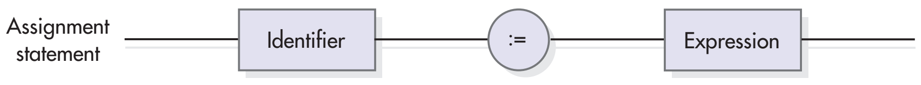


Figure 5.5 Syntax structure charts represent language structure as a diagram.

Syntax structure diagrams show the allowable structure of a language element by tracing a path from left to right; the allowable elements correspond to the 'stations' along the line. Branches are shown as railway 'points', and the path can move onto a branch only by travelling onto the points as if it were a train (see Figure 5.6). Once on a branch, that line must be followed until it reaches the main line again. There will be only one beginning to a diagram and only one ending.

The two text-based metalanguages BNF and EBNF use particular groupings of characters to show the structure of the language.

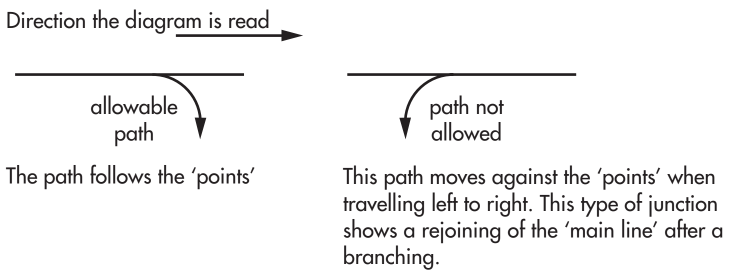

Direction the diagram is read
allowable path
The path follows the 'points'
path not allowed
This path moves against the 'points' when travelling left to right. This type of junction shows a rejoining of the 'main line' after a branching.

Figure 5.6 Reading branches on a syntax structure diagram.

Language elements in BNF are referred to either by their names (if they have been pre-defined) or as individual characters. Those elements that are defined elsewhere (non-terminal symbols) are enclosed by less than and greater than symbols: '<language element>'. Individual characters or strings of characters (terminal symbols) are not enclosed. For example, in BNF …for… means that the word 'for' in a definition is a string of characters. (Note that each of the characters in the word is a terminal symbol.) If it appears in a definition as …<for>…, the word 'for' represents something defined elsewhere. (The word for is, therefore, a non-terminal symbol.)

Alternative elements are indicated by placing a vertical bar in the spaces between each of the alternatives: ' | '. The symbol set ' : : =' is used to represent the statement 'is defined as'. Note that, for clarity, in this text extra spaces have been placed between language elements in both BNF and EBNF; this may not always be the case.

EBNF overcomes a number of problems in BNF, the main one being the inability of BNF to show repetition in a simple manner. In EBNF repetition of elements is indicated by enclosing those elements to be repeated in braces: {*these are repeated elements*}. EBNF also allows for the inclusion of optional language elements (for example an optional + sign in a positive number). Optional elements are enclosed in square brackets: [*this is optional*]. EBNF also allows groups of elements to be indicated. When elements are to be grouped they are placed inside parentheses (*these are grouped elements*). It is interesting to note that, in the case of repetition, a repetition of zero times is also allowed, which makes these elements also optional. The only other difference between EBNF and BNF is the use of the equals sign alone to represent the phrase 'is defined as'.

For basic exercises in the use of these methods of syntax description, refer to Chapter 5 of the Preliminary text.

Commands and their uses

When you code an algorithm in the language you have chosen, you will use a number of features and structures that cannot be easily represented in the algorithm. Each of the languages has its own way of representing these items and the appropriate syntax can be found from the language syntax description. We will look at the ways in which some of these structures can be implemented in a language.

Multi-dimensional arrays

In the Preliminary Course we saw how using a one-dimensional array allows us to store and access a number of data items of the same type by using the one identifier. In practice, however, a single-dimensional array is of limited use. For example, if a business wishes to track the value of each month's sales over a number of years, it is not sensible to use a separate array for each year's results. The best solution to this problem is to create a two-dimensional array that uses two indices. Thus all the sales results can be accessed from the same array, rather than having a separate variable for each of the years' results.

When implementing such an array in our programs, we are faced with the problem of making sure that the right array element is accessed. If more dimensions are needed, such as the monthly sales results from a number of different stores over a period of years, the problem becomes harder. In order to ensure that the right array element is reached, it is best to name each of the indices with a different identifier. So, for example, we could think of an element in the sales figures array discussed above as being represented by the identifier *sales(store_number, month_number, year_number)*. A physical representation such as that in Figure 5.7 makes this representation clearer.

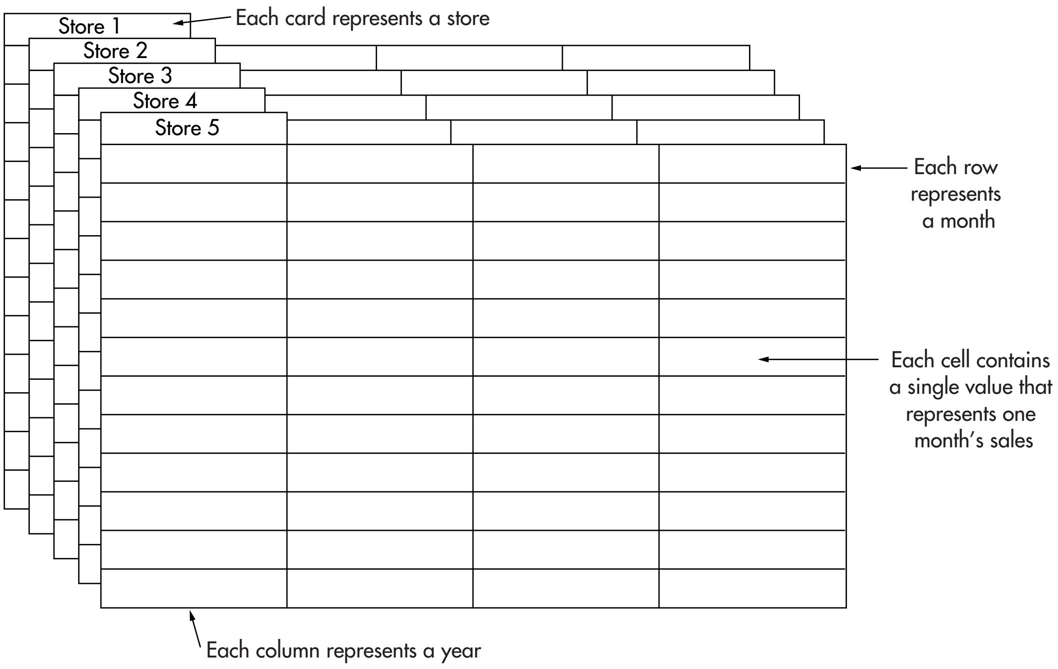

Store 1
Each card represents a store
Store 2
Store 3
Store 4
Store 5
Each row represents a month
Each cell contains a single value that represents one month's sales
Each column represents a year

Figure 5.7 A three-dimensional array can be pictured as a number of cards, each containing a grid of cells, each one holding a data item.

When implementing an array such as the sales figures array in a language, the number of dimensions and the size of the array will usually have to be declared. The choice of programming language will determine exactly how this is done. Dimensioning an array allows the translator to set aside enough memory to store that array. In BASIC, using the DIM statement does this. In Pascal, an array is declared in the TYPE section of the declarations, then the variable is declared as a variable of that type.

The BASIC declaration is:

```
DIM SALES(5,4,12)
```

This declares a variable called SALES which consists of 5 'cards', each with 4 'columns'and 12 'rows'. The terms 'cards', 'columns' and 'rows' have been chosen so that they correspond to Figure 5.7.

The Pascal declaration is:

```
type
   quantity = array[. 1..5, 1..4, 1..12 .] of real;
var
   sales : quantity;
```

The size of the array is set in the TYPE statement, as is the data type to be stored in the array. The variable that we will use is declared as being of the type previously declared. The variable is accessed in the program by using the name declared in the VAR statement.

When a multi-dimensional array is accessed to perform some form of processing, it is important to use the indices in the correct order; otherwise, an incorrect data item will be accessed or an error will occur due to one of the indices being out of the declared range. The following examples, based on the sales array described above, will illustrate how the order in which the indices are changed affects the outcome of the program. These algorithms can be coded as part of a program. The basic algorithm can be modified to suit a wide variety of problems involving arrays needing more than one dimension.

Algorithm 1

The purpose of this algorithm is to process the array elements so that each shop's results for a particular year are dealt with in the order of the months. The array *sales(shop, year, month)* has already been defined and the data items have all been entered. The array is indexed from 1 to 5 for shop, from 1 to 4 for year and from 1 to 12 for month.

```
BEGIN process_each_shop
    set shop to 1
    set year to 1
    set month to 1
    WHILE shop <=5
        WHILE year <= 4
            WHILE month <= 12
                process sales( shop, year, month)
                set month to month + 1
            ENDWHILE
            set month to 1
            set year to year + 1
        ENDWHILE
        set year to 1
        set shop to shop + 1
    ENDWHILE
END process_each_shop
```

An application of algorithm 1 may be to enter the data into the array by using the appropriate input statement as the subprogram *process sales(shop, year, month)*.

Algorithm 2

The purpose of this algorithm is to process the data first by year, then by month, then by shop. This algorithm could be used to calculate the average sales for each store month by month over the four-year period.

```
BEGIN process_each_month
    set shop to 1
    set year to 1
    set month to 1
    WHILE year <= 4
        WHILE month <= 12
            WHILE shop <=5
                process sales(shop, year, month)
                set shop to shop + 1
            ENDWHILE
            set shop to 1
            set month to month + 1
        ENDWHILE
        set month to 1
        set year to year + 1
    ENDWHILE
END process_each_month
```

Note how similar the two algorithms are in their structure. This similarity can cause problems when programmers do not put the loops in the right place.

The sample algorithms can be coded in a language such as BASIC or Pascal and tested. When using a programming language to implement your algorithm, you should ensure that the structure of the coded algorithm matches that of the design as closely as possible. If necessary, you should design the algorithm using those commands and structures that are available in the language. For example, early versions of BASIC did not have a CASEWHERE command, so algorithms designed for implementation in these versions were based on a number of nested IF statements that performed the same task.

Arrays of records

A record by itself is not a very useful data structure. This is because the record structure can hold related data items about only one thing. Usually we need the same data identifier for a number of different things. Rather than use a different identifier for each of the records, it is more efficient to use a single identifier name and to store these records as an array. Remember that records can be stored as an array since they are all of the same data type.

When defining a record structure in a programming language, we need to specify the data type of each of the fields that go to make up the record as well as their names. This information is again needed by the translation system so that the appropriate amount of memory can be set aside for the data.

The example below, although in Pascal, illustrates the declaration of an array of records.

```
TYPE
    article =
        RECORD
            colour : char;
            code, quantity : integer;
            price : real
        END;
VAR
    stock_list : array[. 1..250 .] of article;
```

In this declaration, four fields are defined as making up the record data type called *article*. These are colour, code, quantity and price. The variable to be used within the program is then defined as being an array containing records of type *article*. This array is indexed from 1 to 250, allowing up to 250 records to be stored in the array.

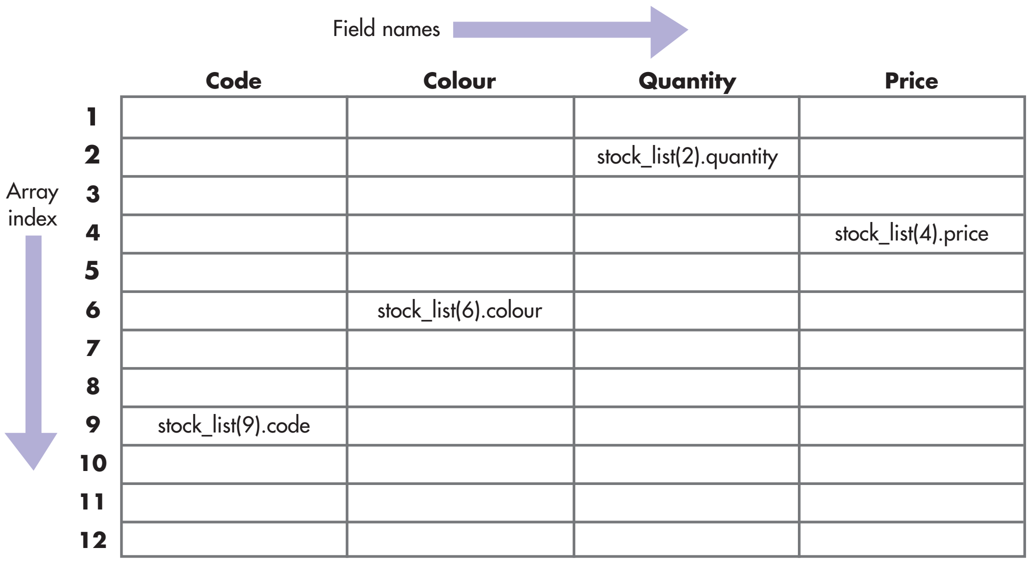


Figure 5.8 A data item within an array of records is indexed using the record's index together with the field name.

When accessing a field of a particular record for processing, both the index and the field name have to be specified. The structure varies little between languages, so the Pascal array defined above will be used as an example. The data item *stock_list(.49.).price* would access the price of the item whose details are stored in record number 49 of the array.

The following Pascal code sample, again based on the array of the example, illustrates how each of the stock numbers and the quantity of each item in stock can be displayed. This is not a full Pascal program.

```
index :=1;
REPEAT
    writeln(stock_list[.index.].code,stock_list[.index.].quantity);
    index:=index+1
UNTIL index > 250;
```

Files (sequential and relative/random)

Processing of a file may involve each of the records in turn or it may involve only those records that meet certain criteria. If all records of a file have to be processed, it is most sensible to start from the first in the file and process each one, in turn, to the last. This is called **sequential processing**. If records that meet one or more criteria have to be accessed, it makes more sense to go straight to the record or records that meet the requirements and just process them. This type of processing is known as **random processing**. When processing the weekly payroll, a company must ensure that every employee is paid; thus the program would access the employee records sequentially. On the other hand, when a bank's customer uses a card to access their account at an ATM, the computer program needs to access just the customer's account details without accessing anyone else's details first. So the ATM program uses random processing when accessing the customer's details.

It is possible for both types of processing to be carried out with the one file and for the one task. For example, if you have an address book program that orders the records by name, in order to find the person whose birthday is 1 April, the program would have to search through the records from the start to find a birthday that matches. This is an example of sequential processing as the records are accessed in order. Once the records containing the birthday 1 April had been found, the other details of the records can be accessed directly that is, by means of random processing.

Algorithm 3

Sequential access will always employ a counting loop to access each record in turn. The counter is also used as the record index. For example, the following algorithm displays each of the marks of a one-dimensional array of marks in turn. The array is already available to this algorithm and contains elements indexed from 1 to *last*.

```
BEGIN display_marks
    set counter to 1
    WHILE counter < = last
        display mark[counter]
        set counter to counter + 1
    ENDWHILE
END display_marks
```

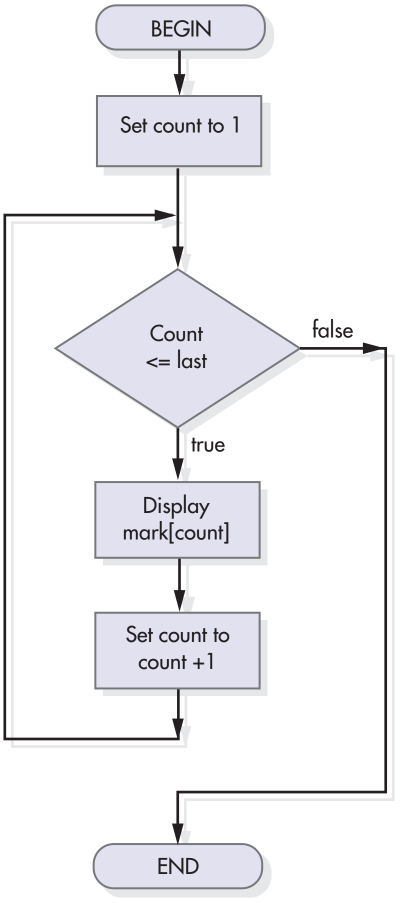


Figure 5.9 Algorithm 3 expressed as a flowchart.

Algorithm 4

Direct access may not involve a loop. In the following example the teacher requires a single mark. In this case the index of the mark is input by the teacher (it may be a student number, for example), the appropriate mark being accessed directly from the array.

```
BEGIN find_student_mark
    set student_number to user input
    display mark[student_number]
END find_student_mark
```

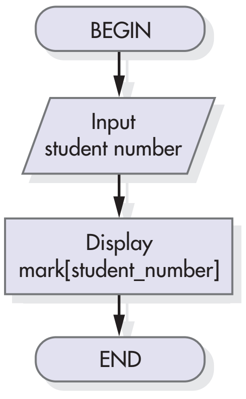


Figure 5.10 Algorithm 4 expressed as a flowchart.

Random number generator

Computer technology is widely used to simulate situations. It is not uncommon for these situations to have an element of chance. Computer languages allow us to bring chance into a program by using a random number generator. A random number generator is a small function that provides a different value each time it is run. Most random number generators will return (that is, 'make') a decimal value between 0 and 1. Some random number functions will allow you to generate values in other ranges. For the purpose of this section, it will be assumed that the value returned is between 0 and 1.

The following sample algorithms illustrate how a random number generator can be used to simulate real situations. In the first example a coin toss is simulated. The output of the program will be either 'heads' or 'tails'. In the second example, the weather in the month of September when the probability of a fine day is 70% is simulated. In this case it will be output each day whether it is 'fine' or 'wet'. The September weather algorithm will also be implemented as a spreadsheet.

Algorithm 5: Tossing a coin simulation

```
BEGIN coin_toss
    set coin to random number
    IF coin <= 0.5
    THEN
        display 'heads'
    ELSE
        display 'tails'
    ENDIF
END coin_toss
```

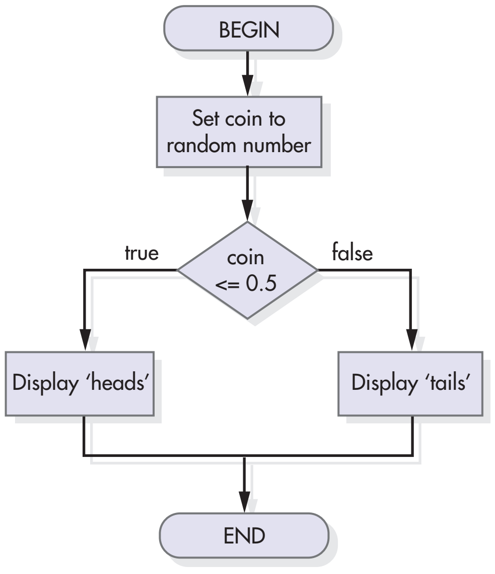


Figure 5.11 Algorithm 5 as a flowchart

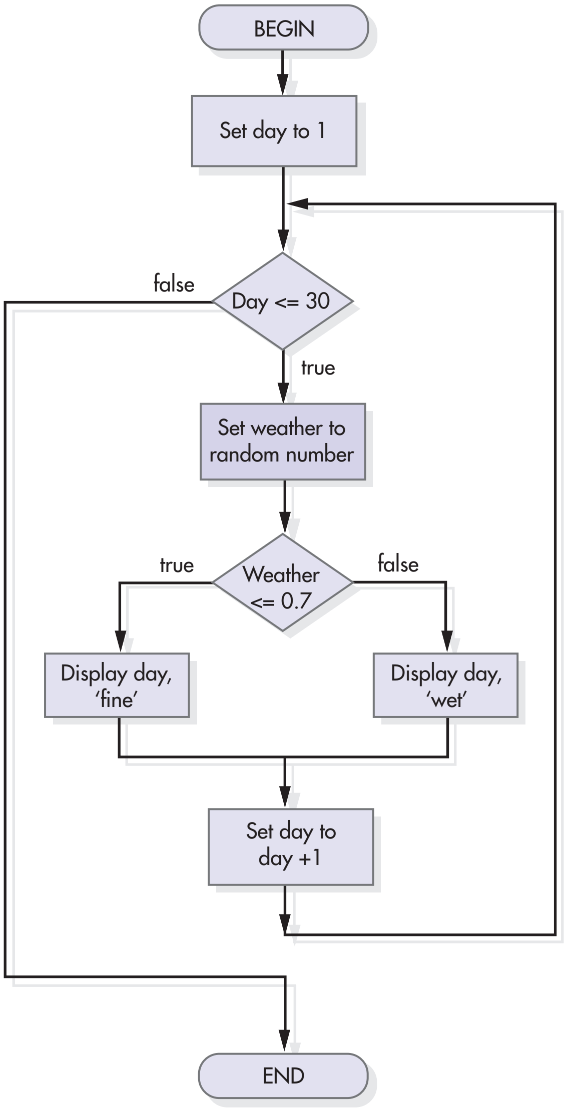


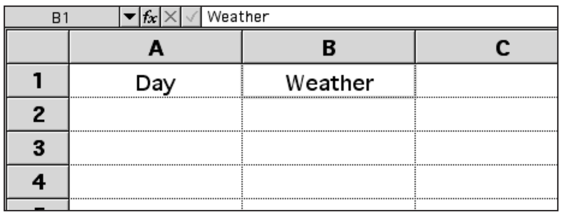
Figure 5.12 Algorithm 6 as a flowchart

Algorithm 6: September weather simulation

```
BEGIN september_weather
    set day to 1
    WHILE day <= 30
        set weather to random
number
        IF weather <= 0.7
        THEN
            display 'September ',
day, ' fine '
        ELSE
            display 'September ',
day, ' wet '
        ENDIF
        set day to day + 1
    ENDWHILE
END september_weather
```

Most spreadsheet applications contain a random number function. It can be used to create simulations quite quickly. Using the September weather simulation as an example, we can see how this is done.

The first step is to place labels at the top of columns A and B. The first label is the word 'day', and the second the word 'weather'. See Figure 5.13.



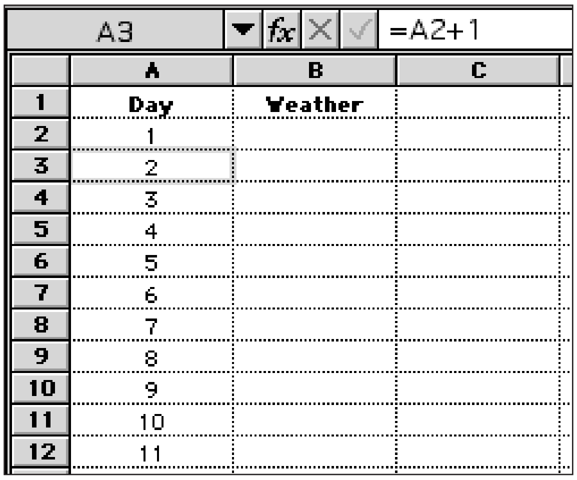
B1 | Weather

	A	B	C
1	Day	Weather	
2			
3			
4			

Figure 5.13 The column headings are placed in the spreadsheet first

The second step in creating the spreadsheet is to place the value '1' in the first cell below the 'day' heading and type the formula *'=A2+1'* in the cell below that. Copy the formula down from cell A3, using the 'Fill down' command. You should see the numbers 1 through to 30 appear in column A, as in Figure 5.14.

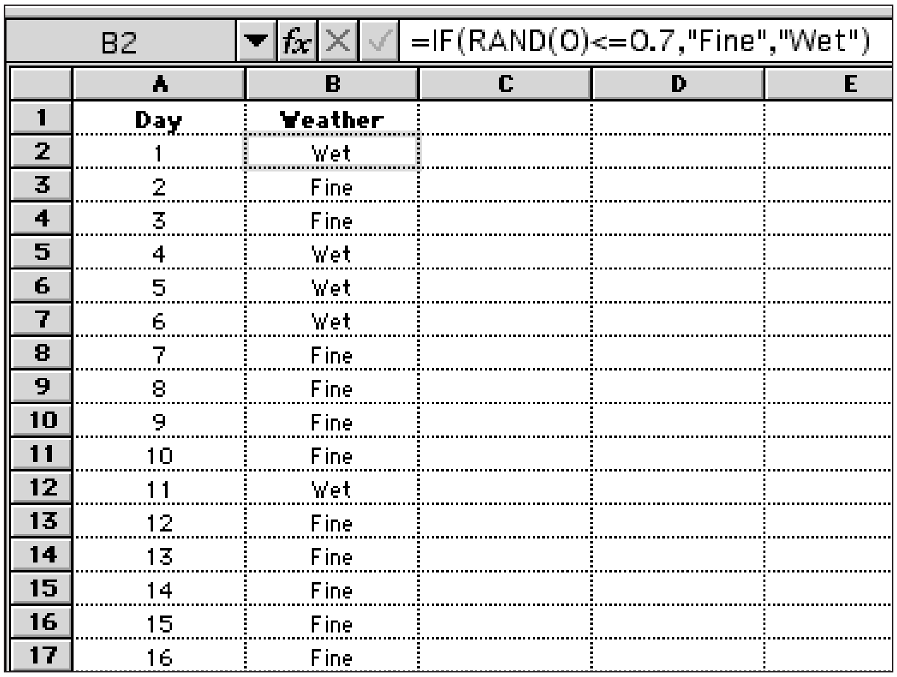
Finally the formula *'=IF(RAND(0)<=0.7, "Fine","Wet")'* is typed into cell B2. (The exact formula may be different for your spreadsheet.) This formula needs to be copied down to cell B31. You will now have a spreadsheet that looks similar to Figure 5.15. Each time you force the spreadsheet to recalculate, the weather pattern will change, as the random numbers generated each time will be different.



A3 =A2+1

	A	B	C
1	Day	Weather	
2	1		
3	2		
4	3		
5	4		
6	5		
7	6		
8	7		
9	8		
10	9		
11	10		
12	11		

Figure 5.14 The days are filled in like this down to cell A31.



B2 =IF(RAND(0)<=0.7,"Fine","Wet")

	A	B	C	D	E
1	Day	Weather			
2	1	Wet			
3	2	Fine			
4	3	Fine			
5	4	Wet			
6	5	Wet			
7	6	Wet			
8	7	Fine			
9	8	Fine			
10	9	Fine			
11	10	Fine			
12	11	Wet			
13	12	Fine			
14	13	Fine			
15	14	Fine			
16	15	Fine			
17	16	Fine			

Figure 5.15 The completed weather simulation spreadsheet.

Exercise 5.2

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

The rules of a language are called the ____________ of that language. Three different methods are used to show these rules. The two text-based methods are ____________ and ____________. The rules can also be shown in a graphical form, using ____________ structure charts (also known as ____________ railroad charts). A programmer can check the ____________ of a statement against the description to ensure that it can be ____________ by the computer.

2 A language uses the following EBNF definition of a variable and an array element. Write a legal two-dimensional array element in this language.

```
character = l | m | n | t
variable =V <character> {<character>}
array element = A <variable> INDEX! <variable> { , <variable> }!
```

3 A multiway selection in the language of question 2 has the following structure.

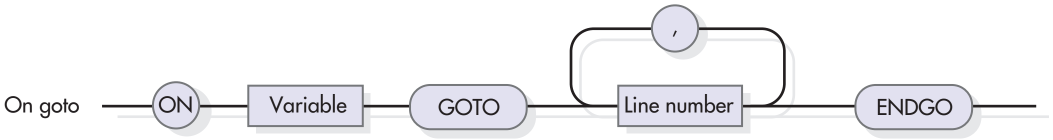


Figure 5.16

Create a legal multiway selection that branches to statements 100, 200, 300 or 400 when the value of *t* is 1, 2, 3 or 4 (this will require the 4 line numbers in the order given).

4 Write an algorithm that finds the longest side of a right-angled triangle using Pythagoras' Theorem. Code this algorithm in Pascal using the syntax structure diagrams in the Appendix of the Preliminary text.

5 A language implements a pre-test repetition using the following definition:

The loop starts with the keyword 'WHILE' followed by a comparison (defined elsewhere). After the comparison is the keyword 'LOOP' followed by a list of statements (defined elsewhere) separated by colons (:). The keyword 'WEND' must finish the repetition. There must be at least one statement after the keyword 'LOOP'. The last statement in the list is not followed by a colon.

6 Using the algorithms in Chapter 4 as a guide, design, code and test a program that inputs ten characters into an array (letters should be used for the test data). This array is then sorted using a bubble sort. Once the array has been sorted, the user can search for a particular character in the sorted list using a binary search, the result of the search being displayed as either the position of the wanted item or the message that the item is not in the list. The process of searching is to continue until the character '@' is entered.

The role of the CPU in the operation of software

The aim of software development is to produce a set of instructions that can be executed by the CPU. Although it is unusual these days to produce a program written in machine code, a knowledge of the manner in which these instructions are expressed helps us to understand how programs actually work.

In the Preliminary Course the fetch–execute cycle was discussed. This cycle describes the manner in which the processor works. That is, it fetches the instruction from primary storage, decodes the instruction, executes the instruction and stores the result. This section looks at the way in which a CPU is arranged and then discusses the way in which the CPU operates.

Since CPUs have different architectures (different designs), we will look at a generic CPU. Also, to simplify the explanation, the sample CPU will not be as complex as those found in modern computers.

All processors contain a number of different parts, each with its particular purpose. The following description is of a CPU with the minimum number of parts. The **control unit** coordinates the actions of the processor. The **arithmetic and logical unit** (ALU) is responsible for all the arithmetic and logical operations carried out by the processor. A number of **registers** (memories) are provided to store results, locations of instructions and flags (bits that indicate the result of an operation). These parts of the processor are joined by means of an internal data bus (a 'roadway' for data to pass along). The processor is itself joined to the primary storage by an external data bus. The address bus that carries the address of the memory location to be accessed by the CPU provides a further link to primary storage. The processor also contains one or more special registers known as accumulators. These features are discussed in more detail in the next section.

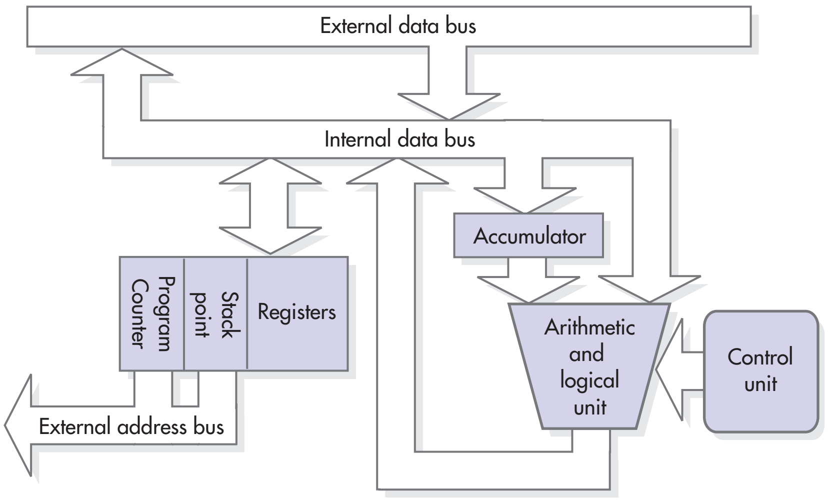


Figure 5.17 Processors are composed of a number of interconnected parts, each with its own purpose.

Machine code and CPU operation

This section looks briefly at the manner in which machine code is structured and operates. This will give us a better understanding of how processes are carried out and their limitations. It will also help should the need arise to write machine code or assembly code. Since there is no single machine code for all processors, generic examples are used.

The sample code used is based on an 8-bit processor, that is, one that can process a single byte at a time. This will allow us to concentrate on the principles of machine code. Current processors can deal with upwards of 4 bytes at a time, with some being able to process 16 bytes. The amount of data that can be processed at one time is described by the size of the **word**. Thus, our sample processor has a word size of 8 bits.

Instruction format

Machine instructions need to convey a number of pieces of information to the processor. This means that the way in which the bits of the instruction are arranged is important. The first few bits are used to 'tell' the CPU the type of instruction to be carried out. These first bits are represented by the most significant bits of the first instruction byte.

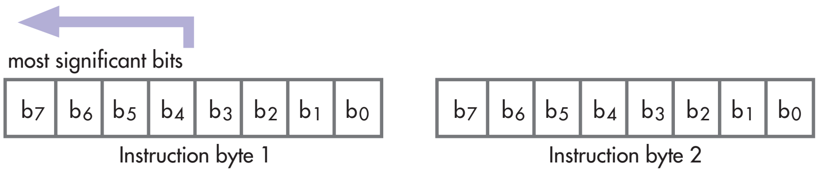


Figure 5.18 An instruction will contain one or more bytes. The most significant bits of the first byte identify the instruction to be performed.

The remainder of the instruction's bits are used to 'tell' the processor the registers to use and the way in which data is to be obtained. For example, the machine code instruction 0111 0011 0000 1101 may represent the instruction 'copy the value 0000 1101 into the accumulator'. The first four bits (0111) represent 'copy'. The second four bits represent 'the value that follows into the accumulator' and the last eight bits are the value to be loaded. All instructions that represent copying a value will start with the four bits '0111'.

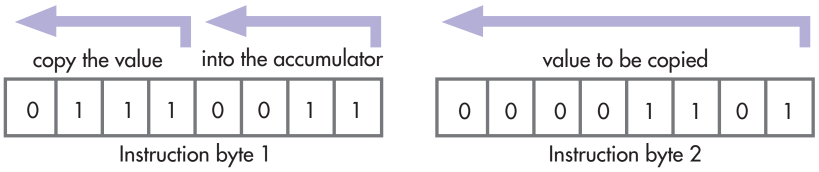


Figure 5.19 The instruction represents 'copy the value into the accumulator'.

Some instructions in this example instruction set will take more than two bytes. For example, if the processor is to access a value in primary storage, the location of the data item needs to be specified. The instruction 0111 0001 0000 1010 1110 0110 represents 'copy into the accumulator the value stored in memory location 0000 1010 1110 0110'. In this case, the first 4 bits still represent 'copy'. However, the next four bits represents 'into the accumulator the value stored in the location that follows', with the last two bytes representing the memory location that supplies the data. This type of instruction is an example of **direct addressing** of a memory location. (Direct addressing modes 'tell' the processor to go directly to a memory location to read or write data.)

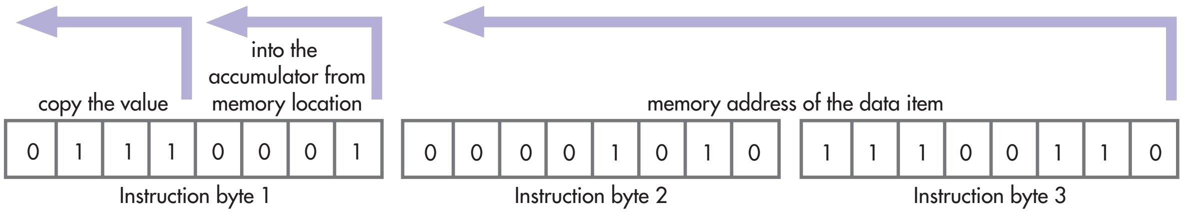


Figure 5.20 This instruction 'tells' the processor to obtain a value from a memory location that follows.

Processor instructions can be grouped into sets that contain similar types. Certain instruction sets are necessary for all processors to function. These instructions include arithmetic instructions such as adding values, subtraction, incrementing (adding one to a value) and decrementing (subtracting one from a value). In addition to arithmetic functions, others are needed for branching (decision making). Branching instructions include unconditional and conditional branches. Comparison instructions are also required. These instructions will output a flag according to the result of the comparison; for example, a flag may be set if two data values are equal. A further set of instructions that handles the use of subprograms is also necessary. The types of instruction mentioned in this section are generally available to all processors. Many processors, especially the more complex ones used in personal computers, will have further instruction sets available for specific purposes.

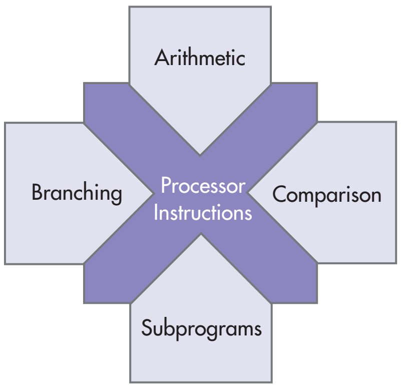


Figure 5.21 All processor instructions can be grouped into sets of similar instructions.

One of the more successful early microprocessors was the Motorola 6502, used in the Apple II and Commodore 64 range of computers. Figure 5.22 shows the types of instruction available for this processor. These instructions are tabulated according to their assembler mnemonics to make the structure clear. Figure 5.23 shows the different addressing modes and operands (the data being processed) available to the programmer using this processor. Combining the instructions with the addressing modes and operands provides a large number of combinations. Current processors may have well over 1000 individual instructions.

Mnemonic	Meaning
ADC	Add with carry, i.e. add a byte, plus the carry flag, to the accumulator
BCC	Branch if carry clear
BCS	Branch if carry set
BEQ	Branch if equal
BMI	Branch if minus
BNE	Branch if not equal
BPL	Branch if plus
BVC	Branch if overflow clear
BVS	Branch if overflow set
CLC	Clear carry flag
CMP	Compare with the accumulator
CPX	Compare with register X
CPY	Compare with register Y
DEC	Decrement (subtract 1 from) memory location
DEX	Decrement (subtract 1 from) X register
DEY	Decrement (subtract 1 from) Y register
INC	Increment (add 1 to) memory location
INX	Increment (add 1 to) X register
INY	Increment (add 1 to) Y register
JMP	Jump to address specified in operand
JSR	Jump to subroutine starting at address specified in operand
LDA	Load accumulator
LDX	Load X register
LDY	Load Y register
RTS	Return from subroutine
SBC	Subtract with carry Subtract from the accumulator and borrow from the carry flag
SEC	Set carry flag
STA	Store accumulator at a certain address
STX	Store X register at a certain address
STY	Store Y register at a certain address
TAX	Transfer accumulator to X register
TAY	Transfer accumulator to Y register
TXA	Transfer X register to accumulator
TYA	Transfer Y register to accumulator

Figure 5.22 The Motorola 6502 instruction set is typical of an 8-bit processor.

Addressing mode	Immediate	Absolute	Zero page	X indexed	Y indexed	Implied	Relative
Operand type	Data	Any address	Page zero address	Address + X register	Address + Y register	None	Offset
ADC	✔	✔	✔	✔	✔		
BCC							✔
BCS							✔
BEQ							✔
BMI							✔
BNE							✔
BPL							✔
BVC							✔
BVS							✔
CLC						✔	
CMP	✔	✔	✔	✔	✔		
CPX	✔	✔	✔				
CPY	✔	✔	✔				
DEC		✔	✔	✔			
DEX						✔	
DEY						✔	
INC		✔	✔	✔			
INX						✔	
INY						✔	
JMP		✔					
JSR		✔					
LDA	✔	✔	✔	✔	✔		
LDX	✔	✔	✔		✔		
LDY	✔	✔	✔	✔			
RTS						✔	
SBC	✔	✔	✔	✔	✔		
SEC						✔	
STA		✔	✔	✔	✔		
STX		✔	✔				
STY		✔	✔				
TAX						✔	
TAY						✔	
TXA						✔	
TYA						✔	

Figure 5.23 The 6502 instruction set is used with various addressing modes and operands to provide an extensive set of individual instructions.

Use of registers and accumulators

As already seen, there are a number of storage locations within the CPU called **registers**. Registers are like the memories on a calculator; they are used to store intermediate results of calculations. Processors will have a number of general-purpose registers and one or more 'working registers', sometimes known as accumulators. The data items that are currently being processed are held in the accumulator. The accumulator is a little like the display of a calculator. (Strictly speaking the calculator display shows the contents of a special memory within the calculator.)

Data that has been requested by the CPU for processing will be stored in the registers and the results passed from a register back to main memory. We will follow the progress of a small machine language program that takes two numbers from two different memory locations, adds them and passes the new value to the first memory location. This is equivalent to the algorithm process *'set sum to sum + value'*.

The machine code instructions would represent the following steps:

```
load the accumulator with the value from memory location 469
add the value stored in memory location 490 to the accumulator
load memory location 469 with the contents of the accumulator.
```

Binary strings representing these instructions would be stored within the main memory. The program can be followed a little better by looking at the values stored in the processor's registers as the program is executed.

In addition to the general registers, the CPU contains a number of special-purpose registers. One of these registers will be used to store special bits known as flags. The flags will be set from zero to one if a particular result has occurred. For example, a bit in the flag register might be set to one if the result of a subtraction is negative. The flag bit can then be tested and a decision made. The next section looks at the use of some of the other special registers in the next section.

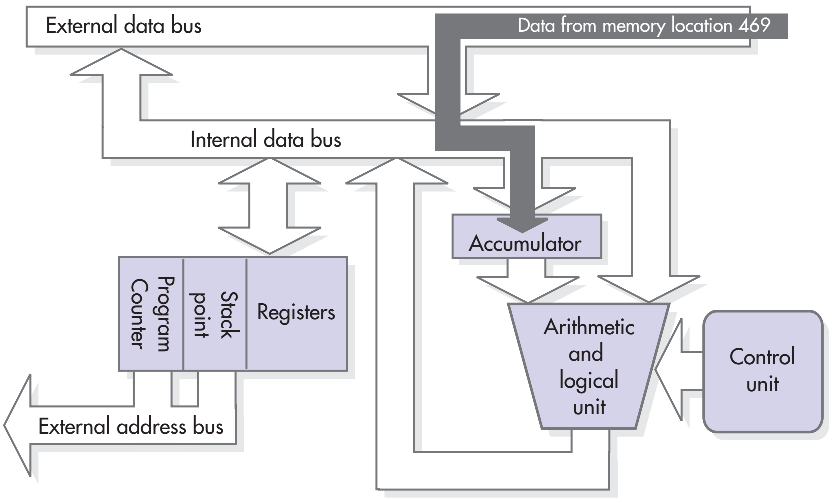


Figure 5.24 The value from memory location 469 is copied to the accumulator.

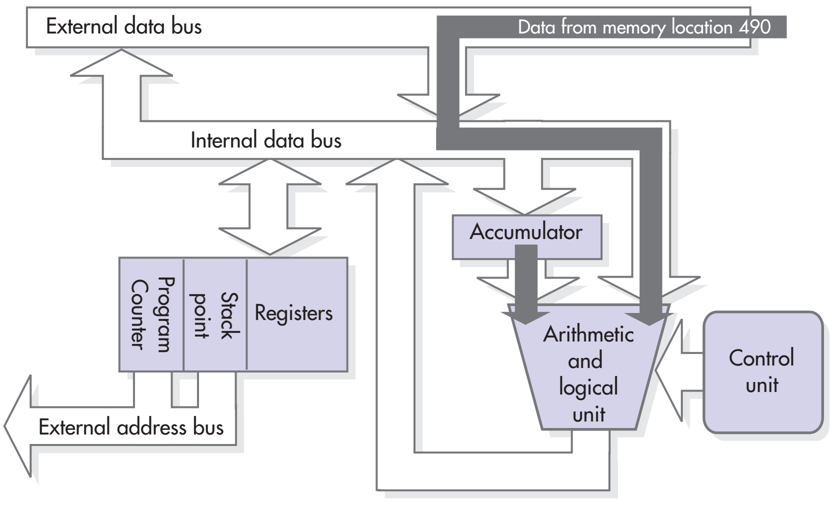


Figure 5.25 The value from memory location 490 is copied to the ALU and the value in the accumulator is copied to the ALU where they are added.

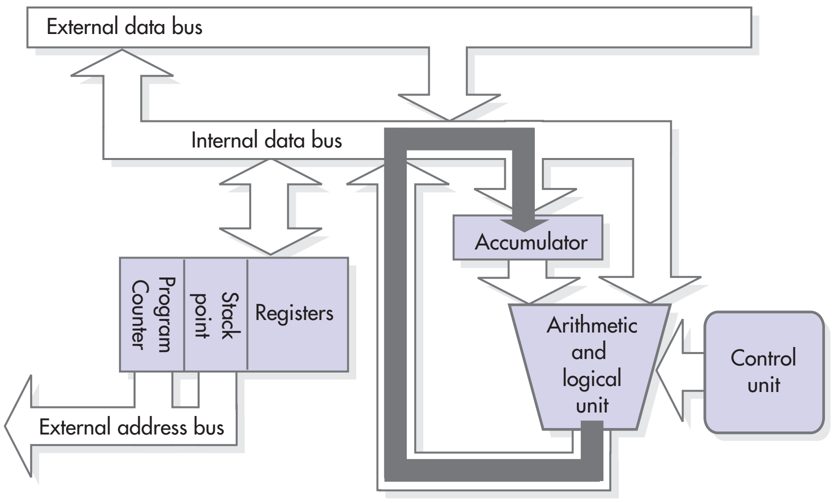


Figure 5.26 The total is copied back to the accumulator.

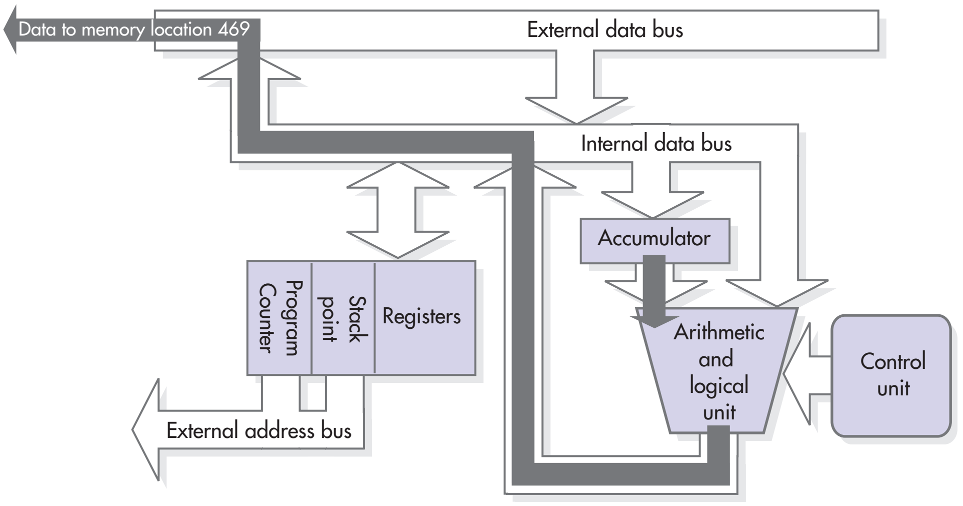
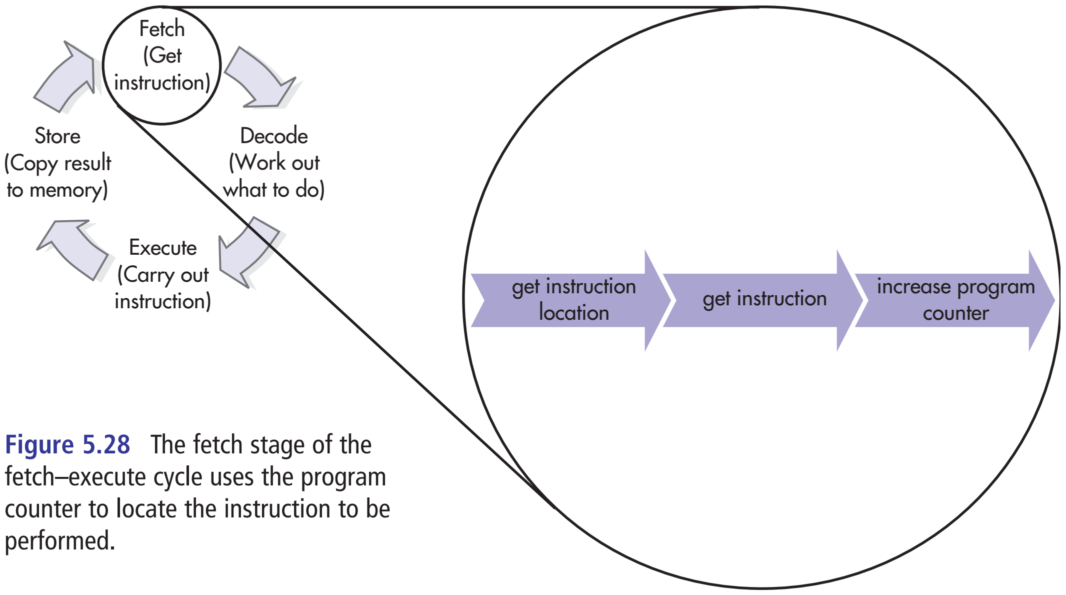


Figure 5.27 The value in the accumulator is copied to memory location 469.

Use of the program counter and the fetch–execute cycle

A register called a **program counter** stores the memory location of the next instruction to be processed. As the 'fetch' stage of the machine cycle is reached, the memory location stored in the program counter is used to locate the next instruction. Once the current instruction is fetched from main memory, the program counter is increased to point to the next instruction. If the program instruction requires control to pass somewhere other than the next instruction, the program counter is adjusted to point to the required instruction.



Fetch (Get instruction)
Decode (Work out what to do)
Execute (Carry out instruction)
Store (Copy result to memory)
get instruction location
get instruction
increase program counter

Figure 5.28 The fetch stage of the fetch–execute cycle uses the program counter to locate the instruction to be performed.

Addresses of called routines

A further register, known as a **stack pointer**, is used to keep track of the location of the beginning of a special part of main memory known as the stack. The stack is used by the CPU to hold the address of the next instruction when a subprogram is run. This allows the CPU to return to the correct place in the main program after the subprogram has finished.

The stack is a special set of locations in the main memory that have been set aside for the use of the CPU. These locations are used to keep track of the places where a program jumps to a subprogram. This allows the program to resume at the right place when the sub-program has been executed. The stack is a special type of list, known as a LIFO (Last In First Out) list. This means that the first value to be removed from the list will be the last one that was placed there. The last location in the stack to have an item placed in it is known as the top of the stack. This data structure is rather like a stack of plates where the last plate put on the stack will be the first one taken off when they are used. The structure obtains its name from this analogy. A clearer understanding of the working of the stack can be obtained if the operation of a subroutine is examined.

In this example the program will branch to the subroutine that begins at memory location 200, execute that routine, and then transfer control back to the main program.

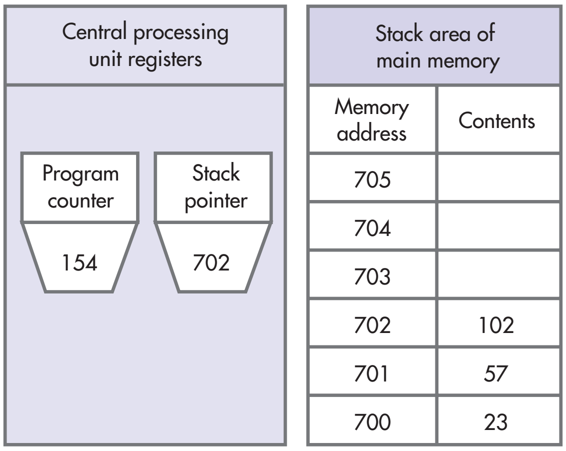


Figure 5.29 The status of the program counter, stack pointer and stack before the instruction 'call the subroutine at memory location 200' is executed.

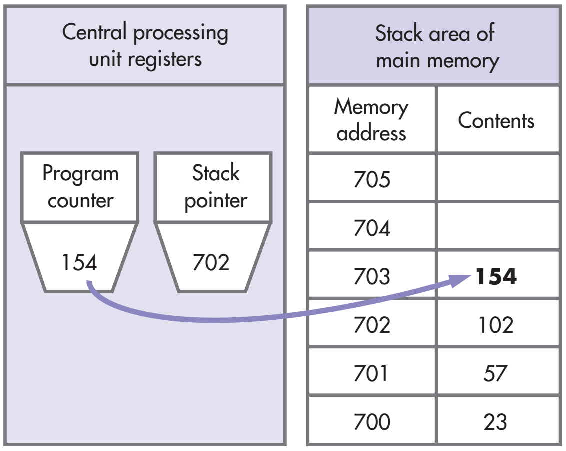


Figure 5.30 The contents of the program counter are first copied to the location at the top of the stack.

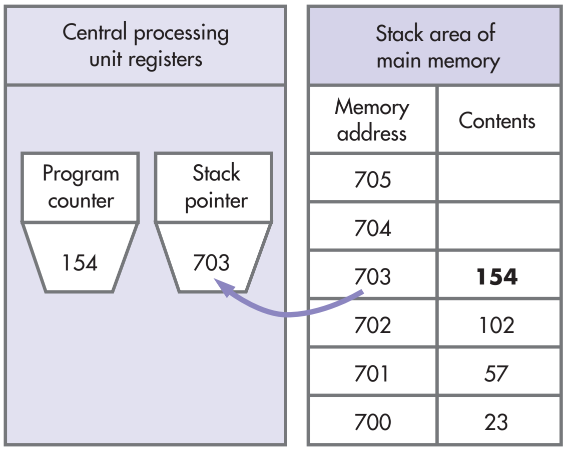


Figure 5.31 The stack pointer is changed to point to the new top of the stack.

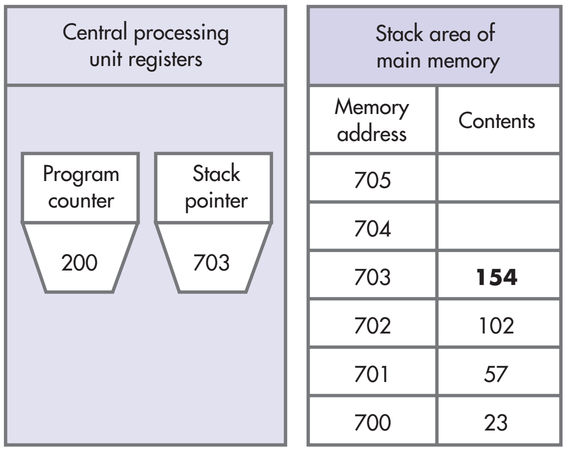


Figure 5.32 The program counter is set to the address of the first instruction in the subprogram.

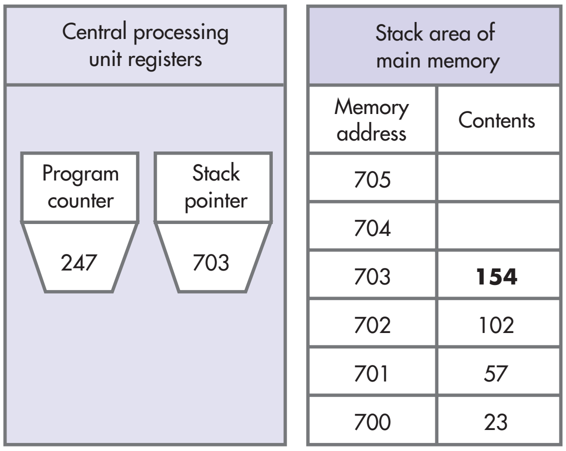


Figure 5.33 At the end of the subroutine this is the status of the program counter, stack register and stack.

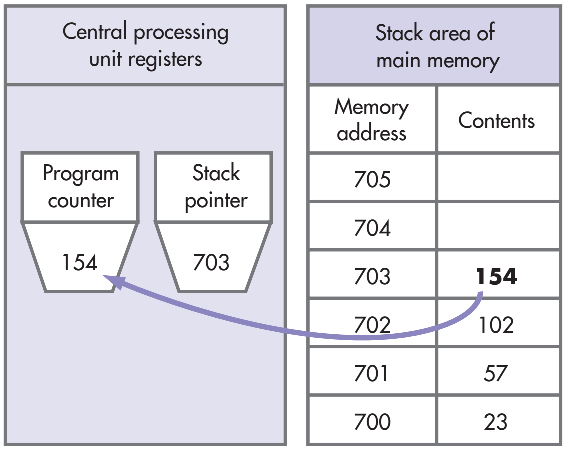


Figure 5.34 The value at the top of the stack is copied back to the program counter.

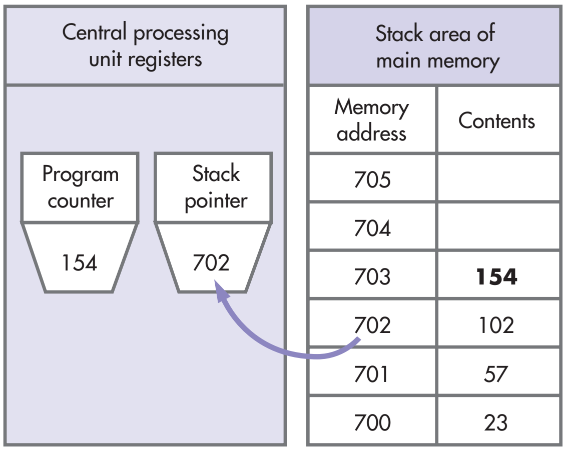


Figure 5.35 The stack pointer is changed to point to the top of the stack one place lower. This effectively removes the address from the stack.

Linking, including use of DLLs

Subprograms used by a main program can come from a number of sources. One of the sources is the programmer who creates a subprogram to perform a task that is specific to the program. A second source of subprograms is the library that comes with the development system, for example the subprogram that calculates the square root of a number. A third source is the set of subprograms that form part of the operating system, for example a subprogram that copies a file from main memory to an external storage device such as a hard disk.

When a program requires a particular subprogram, its address is incorporated into the program by a part of the translation system called a **linker**. The linker places a call to the subroutine into the program. When the program reaches that point in its execution, the subprogram is run and control is then passed back to the program at the end of the subprogram.

A special type of subprogram library, known as a DLL (Dynamic Link Library) is in common use these days. A DLL is used in the same way as any other library, however, the library can be updated with a new one, all programs that use the DLL automatically linking to the new version. This allows for modifications to be made to the library after a program has been written, the changed subprograms are, however, automatically incorporated into the program when it is run.

Exercise 5.3

1 Copy the following passage and complete by filling in the blanks with the appropriate terms or phrases.

A CPU contains a number of sections. These sections are the __________ and logical unit (__________), the __________ unit and the __________ which are used for __________ storage. The __________ is responsible for carrying out all the instructions. The __________ coordinates the actions of the __________. The processor is joined to __________ storage by means of an __________ data bus.

2 Describe the steps in the fetch–execute cycle.

3 Explain the steps that a processor will carry out when it copies a value from memory location 548 to memory location 376. Illustrate your answer with a diagram.

4 Investigate the Internet site of a microprocessor manufacturer such as Intel, IBM or Motorola and download a data sheet for developers that contains the instructions for a particular processor. Choose one of the instructions and write this instruction as a binary string. Identify each of the sections of the instruction. Describe the purpose of the instruction.

5 Using the data sheet you have from question 4, identify and describe the types of register that your processor contains and the purpose of the specialised registers.

6 Describe the way that a processor executes a subprogram and then returns to the appropriate place in the main flow of the program. Use an example to illustrate your answer.

Translation methods in software solutions

Programs written in any of the high-level languages (second-generation and above) cannot be directly understood by a processor. The instructions contained within the program have to be converted from the human readable form in that high-level language (called the source code) into a machine-readable form (the object code). Source code is usually created using a text editor that forms part of a program development system. However, a text file that has been created by other means, for example in a word processor, may also be translated. The process of translation from the source code to the object code is accomplished by using one of three translation methods: compilation, incremental compilation or interpretation.

Different methods of translation

Compilation

Compilation, using a compiler, involves translating the whole of the source code into object code and storing the object code to be executed later. Compilation is similar to the translation of a book from one language to another, in which the whole book is translated before being read. A compiled program will be executed quickly, since the computer can understand the instructions directly.

Interpretation

In interpretation the source code is translated line by line into object code which is immediately executed. Interpretation can be likened to an interpreter who stands beside a foreign dignitary and translates the sentences into the known language as they are spoken. Interpreters produce code that is immediately executed and therefore have the advantage of being able to identify errors within a statement at the time of execution. A second advantage of the interpreter over the compiler is that the program can be tested for both syntax and run-time errors. Interpreters were commonly used in the early personal computers such as the Commodore 64 and Apple II, which had a BASIC interpreter in ROM. An interpreted program will be executed more slowly than compiled object code since the translation process has to be carried out each time a program is executed. This is most evident where there are a number of loops within the program. A compromise is to use an interpreter during the development stage of a program and compile the resulting source file when all errors have been identified and corrected.

Incremental compilation

When a program is run using incremental compilation the commonly executed routines are translated into machine code and stored. The program itself is translated using an interpreter, but when these compiled routines are needed they are run from the stored code rather than being retranslated each time the code is reached. This speeds up the process of running the interpreted program while keeping the advantages of an interpreter.

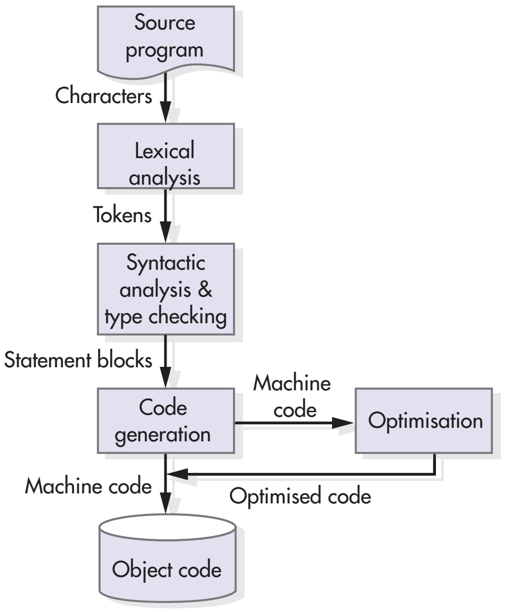


Figure 5.36 The process of translation.

The translation process

The language translator is itself a computer program. Input for this program is the source code that consists of high-level language that has been entered as text. Output from the translation system is the executable object code. The language structure expressed in a form such as a structure diagram or BNF determines the algorithms that are used within the translator.

The three methods use similar processes to translate the source code, differing only in what occurs after a line of code is converted. Interpretation, compilation and incremental compilation all involve the processes of lexical analysis (also called scanning), syntactic analysis (also known as parsing), semantic analysis (or type checking) and code generation. A compiler may also optimise a program in order to make it run more efficiently (see Figure 5.36).

When programs are written in modules, it is quite likely that each module (or segment) will be compiled and tested independently of the others. A linker is used as a means of joining these modules into a single object file. The primary purpose of the linker is to add routines from the operating system and user libraries (such as printing modules and mathematical functions) and also place the appropriate call and return instructions so the program segments are properly joined. The linker also ensures that all blocks of data declared to be common to more than one segment are able to be accessed by all those segments that need them.

We will follow the progress of the following TUSIL program (see the Preliminary text for a description of the language) which has been written to calculate the length of the hypotenuse of a right-angled triangle using Pythagoras' theorem:

```
program PYTHAGORAS:
    declaration!
        SIDE_A is a real number!
        SIDE_B is a real number!
        HYPOTENUSE is a real number!
    end declaration:
    readin('What is the length of the first side ' ! SIDE_A):
    readin('What is the length of the second side ' ! SIDE_B):
    HYPOTENUSE RB √(SIDE_A ^ 2 + SIDE_B ^ 2):
    typeout('The hypotenuse is' ! HYPOTENUSE):
end program%
```

Lexical analysis (scanning)

The first stage in the translation process is that of lexical analysis, in which the source code is read one character at a time. The lexical analyser uses the rules set down by the syntax of the language to create recognisable language elements from the characters of the source code. Characters that form programmer documentation, such as spaces, indentations and comments, are removed during this process as they are of no use to the translator. Once a language element has been identified, it is assigned a code called a token.

The scanner creates two categories of token:

- Tokens representing elements defined as part of the language syntax. The most common of these are reserved words (for example *program*, *declaration*, *readin* and *typeout* from the sample program), constant values (such as the numbers 2 and 3.14159) and operators (for example +, –, * and /).
- Tokens representing elements created by the programmer, usually called identifiers, to represent containers (variables), procedures and the like (for example *SIDE_A* , *SIDE_B* and *HYPOTENUSE* from the example).

The identifier tokens, or labels, are stored in a symbol table or token dictionary, which also lists the names, thus allowing the translator to keep track of them. The symbol table lists other attributes of identifiers, such as its data type if it refers to a variable. The translator uses attributes listed in the symbol table to allocate memory to variables, check for type mismatches and make calls to subprograms. An interpreter, for example, can store the memory location of each variable in this dictionary, allowing it to refer to that location when the variable is encountered later during execution; a compiler will store an address relative to the start of the program. In the previous example the variable declaration provides the translator with the attributes of each variable. The definitions of functions and subprograms also provide information that is kept in the symbol table. In languages such as BASIC this table is added to each time a new variable is encountered during scanning. Some variable types may not be resolved until parsing.

Token identification	Name	Kind	Type
PRO	PYTHAGORAS	Program ID	
VARI	SIDE-A	Variable	Real number
VAR2	SIDE-B	Variable	Real number
VAR3	HYPOTENUSE	Variable	Real number

Figure 5.37 A representation of the symbol table for the TUSIL program.

Symbol	Meaning
□▧	Punctuation token
□▨	Constant token
□▥	Operation token
□□	Identifier token
□▤	Reserved word token

Figure 5.38 Key to the symbols used for the tokens used in Figures 5.41 to 5.43.

The tokens are then passed on to the syntactic analyser that arranges them in a way that can be understood by the computer. Figures 5.39 to 5.41 show an example of code passing through the scanner and being converted to a series of tokens that pass in sequence to the syntactic analyser. Figure 5.38 gives the key to the symbols that are used for the tokens in the example.

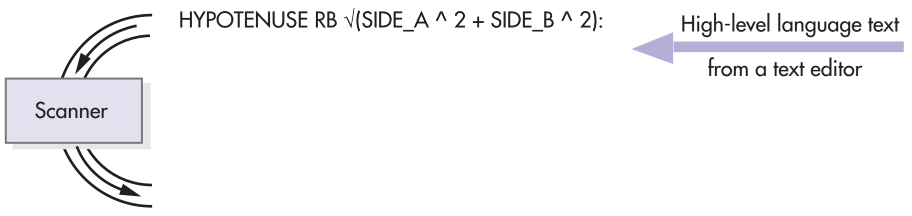


Figure 5.39 The 9th line of high-level code enters the scanner as text.

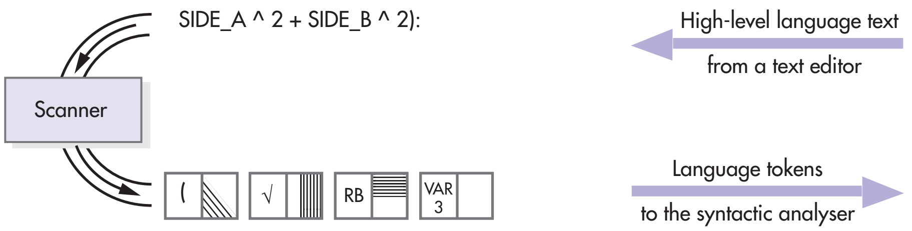


Figure 5.40 As the code passes through the scanner it is converted to a series of tokens which are passed in sequence to the syntactic analyser.

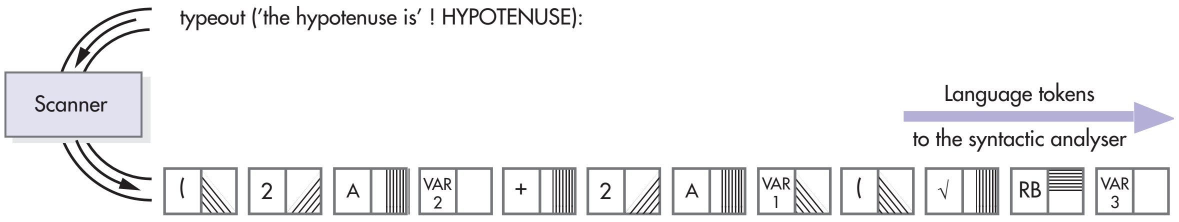

typeout ('the hypotenuse is' ! HYPOTENUSE):
Scanner
Language tokens
to the syntactic analyser
(
2
A
VAR 2
+
2
A
VAR 1
(
√
RB
VAR 3

Figure 5.41 The whole line has now been tokenised and passes on to the next stage of translation. The next program line is prepared for conversion.

Syntactic analysis (parsing)

As the tokens pass from the scanner, they move to the syntactic analyser which arranges them in a way that allows the computer to understand the logic of the program. This arrangement is best viewed as a hierarchical structure called a parse tree.

It was seen in the Preliminary Course and earlier in this chapter that the syntax of a language can be represented by defining various language elements and the relationships between them. Parsing is an implementation of these rules so that the meaning of the program can be determined. The rules are those expressed in the syntax description, and they form the basis for the algorithms that govern the operation of the translator.

The process of parsing to discover the meaning of a sentence can be illustrated by taking a simple English sentence that can be interpreted in two different ways. The sentence 'The boy hit the girl with the ball' has two different meanings. The first is that the boy threw a ball that hit the girl; the second is that the boy hit the girl who was holding the ball. The two different parse trees of Figures 5.42 and 5.43 illustrate these interpretations. Computer languages must be designed so that there is only one parsing, and therefore only one meaning, for each 'sentence'. This is necessary since an instruction given to a computer must have exactly the same meaning, and therefore produce the same actions, each time it is executed.

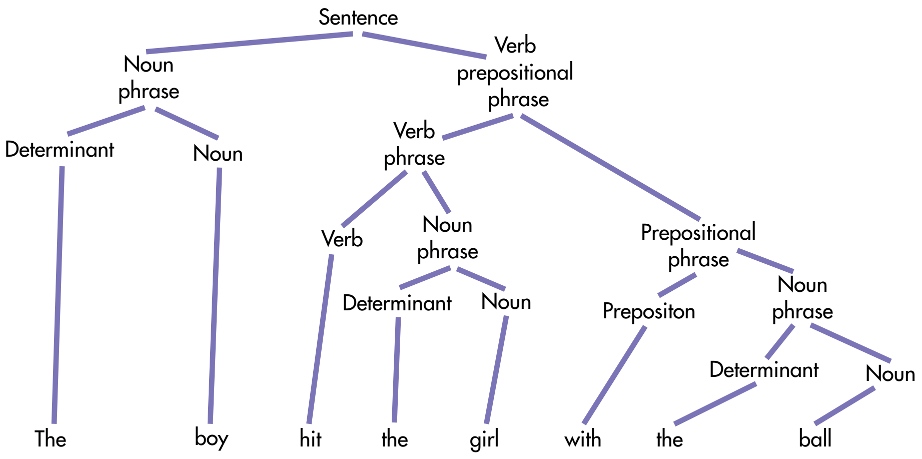


Figure 5.42 Interpretation of the sentence as the boy throwing the ball at the girl and hitting her with it.

The structure of the whole TUSIL sample program is too complex to represent as a single parse tree. From Figures 5.44 and 5.45 the structure of this program can be examined and it can be seen how it conforms to the TUSIL syntax rules in the Preliminary Course.

If a group of tokens does not conform to the rules of the language, the syntactic analyser cannot place them on the parse tree and so assumes that there is an error in the program.

When an error occurs, the translating program reports a message back to the programmer. These error messages may consist of only a numerical code or they may present a more detailed report indicating the location of the error, its type and perhaps a suggestion for its correction.

The syntactic analyser in an interpreter and an incremental compiler will process only a single high-level language instruction before sending the tokens on to the next stage of translation. A compiler will parse the whole program before sending the tokens along to the next stage.

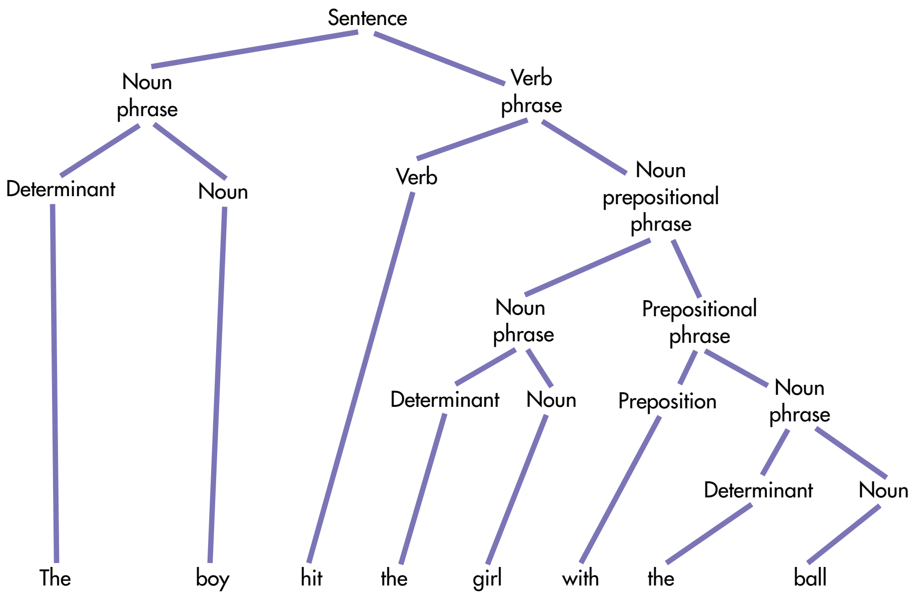


Figure 5.43 Interpretation as the boy hitting the girl who was holding the ball.

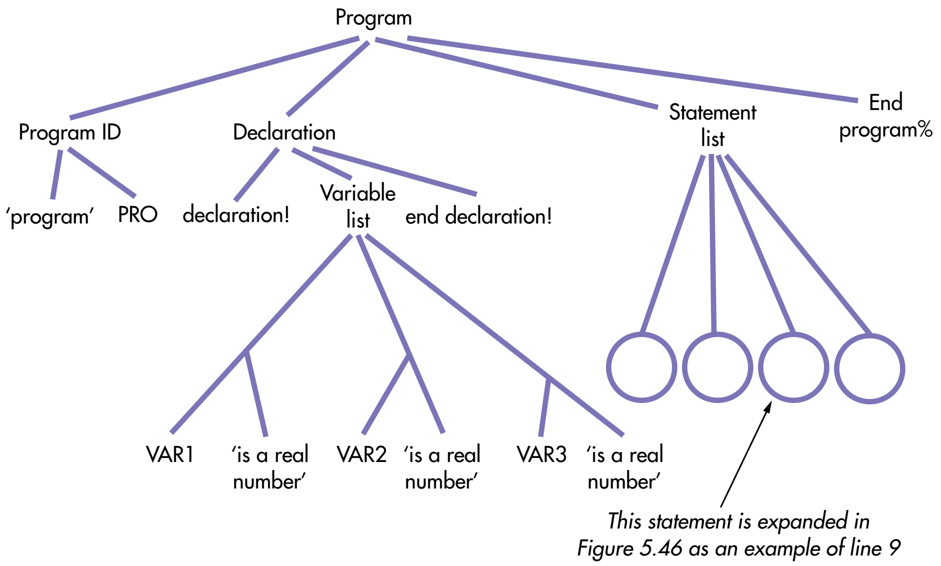


Figure 5.44 A tree representing the structure of the TUSIL program

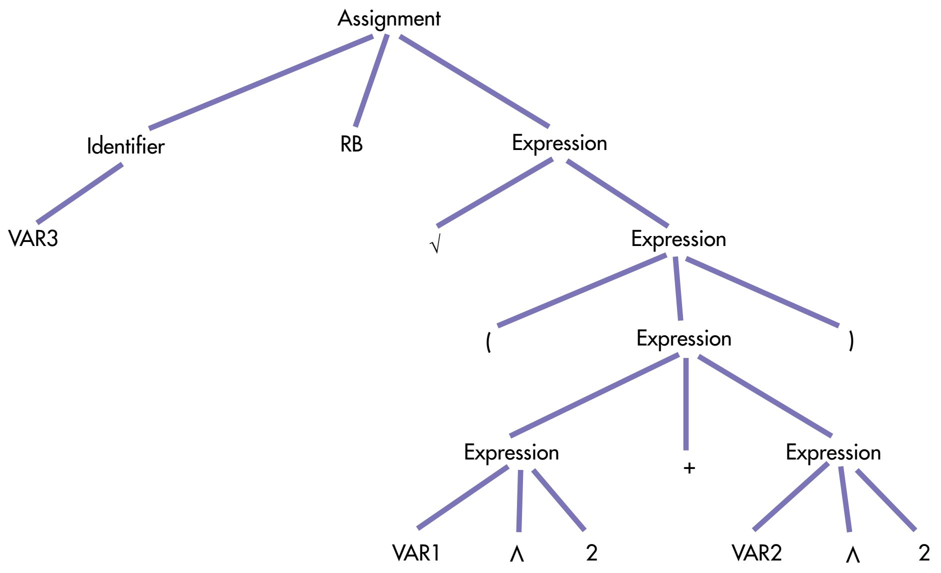


Figure 5.45 A tree representing the structure of line 9 in the TUSIL program.

Type checking

The parsed tokens are sent on to the type checker. This part of the translator has two purposes:

- detecting data types within the tokens and passing that information on to the translator
- detecting incompatible operations between differing data types and creating error messages.

As seen earlier in the course, data can have different types, for example real and integer number types. Variable declaration at the beginning of a program or module will indicate to the interpreter the number of memory locations required to store each data item. Following the allocation of storage locations, the interpreter can refer to these locations during translation each time one of the variable names is used within the program. The type-checker provides the interpreter with information necessary for this task to be carried out. If the language does not employ variable declaration, the type checker examines those variables within each block of tokens using the syntax rules to determine the variable types within that statement.

Another consideration is that operations may be defined differently for different data types. For example, the operation of addition (usually shown by using the + symbol) may be defined differently for string data than for numerical data. If a programmer accidentally tries to add a string to an integer, the interpreter needs a mechanism for detecting it. Further, the result of an operation needs to be stored, but how does one store the sum of a string and an integer? What data type is it? These errors are also detected by the type checker and signalled by means of an error message.

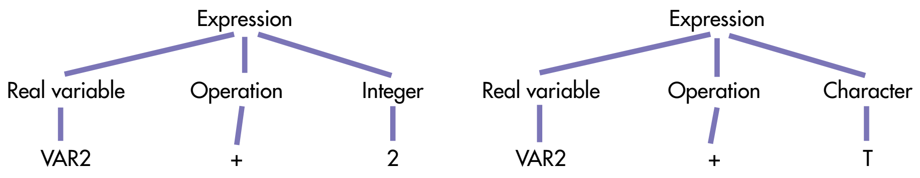


Figure 5.46 Representation of type matching a legal instruction (on the left) and an illegal instruction (on the right) in a parse tree.

The type checker uses the information stored in the symbol table to determine the data type represented by each of the tokens within the operation. These are compared to a list of legal combinations. If the combination is legal, the type checker determines the output type and passes on to the next branch of the tree; if it is not legal, it issues an error message.

Code generation

The code generator performs the task of conversion from tokens to object code, the code that is executable by the computer. The generator traverses the parse tree according to a set of strict rules, creating appropriate machine code whenever it gathers sufficient tokens to form a machine command. It continues through the tree in this manner until all instructions have been coded.

The code generator is also governed by the syntax rules used to write the source code. The code generator used for this example starts at the top left of the tree and moves downwards and to the right around the tree. As it moves around the tree, it chooses tokens from the ends of the branches, moving left to right around the tree. This process is known as traversing the tree.

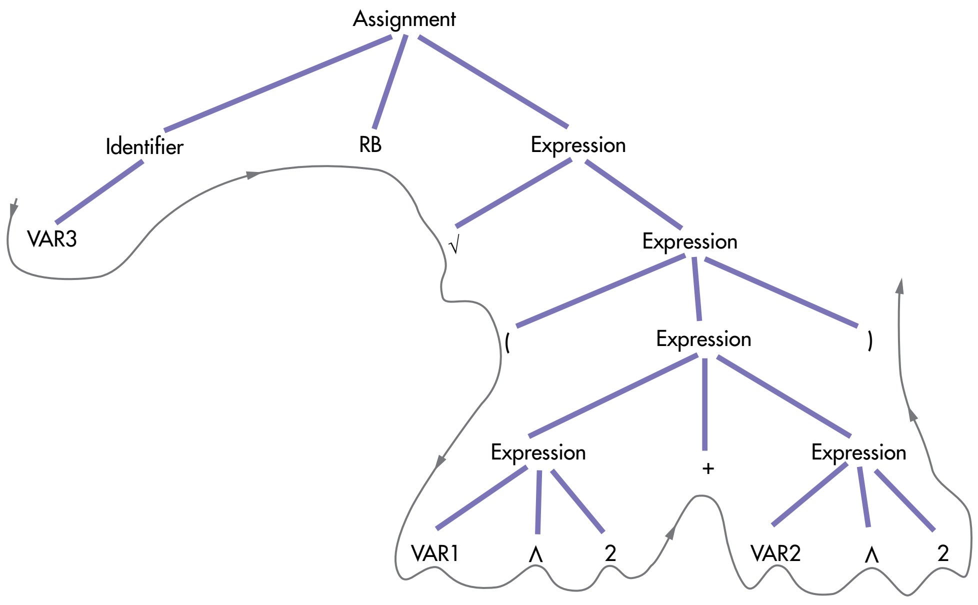


Figure 5.47 Traversing the parse tree for the TUSIL code example used in this section.

The translated code, called object code, may then be used. This code is stored in main memory by a compiler from where it may be stored externally, executed or manipulated. An incremental compiler will add the code generated from each expression in the program to the code that already exists in memory. Interpreters execute the code from one parsed expression before starting the translation process on the next expression.

The linker

Machine code produced by the code generator will often not be in a form that allows it to be executed by the system. Even the simplest programs will require calls to the operating system, for example for input and output. In addition, the modular approach of top-down programming will often produce smaller programs which need to be joined together to make the final solution. This is the task of the linker.

High-level language systems have to be able to provide the programmer with ways of implementing the features of the language. This is accomplished by incorporating a number of pre-written subprograms and functions into the various libraries. These library routines are then made available to the translation system.

A linker will make links to the operating system by putting the address of the required subroutine in the appropriate position in the object code. During execution, when the link is reached, control passes from the user's program to the appropriate subroutine in the operating system, passing back to the user program after execution. Library routines are placed in the object code by the linker and executed as if they had been written by the programmer as part of the program.

The loader

Programs do not necessarily occupy the same memory locations each time they are executed. This is especially true when two or more programs are in main memory at the same time. If the programs were compiled using absolute addressing (that is, each reference to memory specified a particular location), it is very likely that they would both modify, or read, the same memory location, with possibly disastrous results. To overcome this problem, programs are compiled using relative addressing (each referral to a memory location being specified as a number of locations from a reference location) or a loader is added into the code.

When a program is run, the loader places a small utility program, called a load module, in main memory before execution takes place. The purpose of the load module is to adjust all memory location references so they do not interfere with other parts of the system.

The optimiser

A translator, as described above, can work only with the code that is being processed at that time. It cannot review code that has already been translated and it cannot anticipate what code may occur later. This can lead to code that is badly organised or unnecessary (redundant). There is no solution for this problem when an interpreter is used, as the code is immediately executed; however, a compiler may be able to improve the object code by means of an optimiser, as the machine code is all in memory. The aims of an optimiser are to reduce the amount of code so that the object code occupies less memory and to organise the code so that the program runs faster.

Some optimisation methods are very complex and may greatly increase the compilation time. To overcome this problem, some compilers target certain portions of the object code which are likely to be used a large number of times, such as submodules (procedures) and repetitions (loops).

Advantages and disadvantages of each method

Each of the translation methods discussed above has its strengths and weaknesses.

Advantages of a compiler

- Compiled programs will run faster than those that have been interpreted as they are already in a form that the processor understands.
- Compiled programs hide the code from view so that it is more difficult to determine the algorithms used.
- A compiled program is often a lot smaller that the high-level code that generated it.
- A compiled program cannot be easily modified by an inexperienced user.

Disadvantages of a compiler

- Run-time errors are not apparent until the program has been completely compiled.
- When a compiled program is modified, the whole of the program has to be re-compiled, regardless of the nature of the modification. Even if a minor modification is made, the whole program has to be re-compiled. This can make the testing process tedious.

Advantages of an interpreter

- During testing, both run-time and translation errors become apparent as the code is being executed. These errors may then be corrected as they are found. This allows the programmer to more quickly debug a program, as the code does not need to be completely translated after the changes have been made.
- The process of interpretation also allows the programmer to quickly add and remove debugging aids such as flags and debugging output statements.

Disadvantages of an interpreter

- Programs that are interpreted will run far more slowly than those that have been compiled, as each line of code has to be translated before it is executed.
- The code of an interpreted program is easily accessible to any user or other programmer. This means that the illegal use of modules is easier.
- Programs that are interpreted will generally take up more storage than a similar program that has been compiled.

Advantage of an incremental compiler

- Programs will run faster than those of an interpreter; however, the incremental compiler retains the advantage for a programmer that both run-time and syntax errors can be identified as they are reached.

Disadvantages of an incremental compiler

- Programs are not executed as quickly as for a compiled program.
- Code is still accessible to users and others.

Exercise 5.4

1 Complete each of the following statements with the most appropriate word from the list:
blocks, compiler, interpreter, linker, object code, optimiser, syntactic analyser, source code, symbol table, type checker

a A program written in a high-level language is called the ________.

b As a result of compilation, ________ is saved in memory from which it can be run or saved to secondary storage.

c A(n) ________ is often used in a compiler to make the machine code more efficient.

d The ________ will report an error if an attempt to add an integer to a string is encountered during translation.

e A ________ is a group of tokens that represents elements of the program that serve a common purpose.

f A ________ provides information about identifiers to the interpreter.

g Tokens are organised by the ________ into a hierarchical structure according to the language's syntax rules.

h A ________ changes all the source code into machine code before execution can take place.

i Using a(n) ________ is the slowest method of executing a program since statements are translated only as they are needed.

j Programs are often written in modules; a ________ is used to join these pieces together to form a program.

2 Describe the three methods of translating a high-level language into machine code. What are the advantages of each of the methods?

3 What are the four steps that are common to all translation methods? Describe their actions in your own words.

4 Explain, in your own words, the purpose of a linker.

5 In a high-level language of your own choice, write the statement that corresponds to the mathematical expression:

$$\text{Amount} = \text{Principal} \times (1 + \text{Rate}/100)$$

Draw a diagram to illustrate how the parse tree may be constructed for this statement.

6 Construct a symbol table for the following Pascal program. (Remember that all identifiers should appear in the table.)

```
program numberscales (input,output);   {Program declaration}
const       {Start constant declaration}
    maxradix = 10;
    maxlen = 32;
    minus = '-';
    zero = '0';{End constant declaration}
type        {Start type declaration}
    radix = 2 .. maxradix;{End type declaration}
var     {Start global variable declaration}
    datum : integer;
    scale : radix; {End global variable declaration}
procedure writenumber (num : integer; rad : radix);
        {Procedure declaration}
    var     {Start local variable declaration}
        jp , kp : 0..maxlen;
        buffer : array [1..maxlen] of char;
            {End local variable declaration}
begin       {Begin procedure}
    if num < 0
        then
            begin
                write(minus);
                num := abs(num)
            end;
    kp := 0;
    repeat
    kp := kp + 1;
    buffer[kp] := chr(num mod rad + ord(zero));
    num := num div rad
    until num = 0;
    for jp := kp downto 1 do
        write(buffer[jp])
end; {writenumber procedure}     {End procedure}
begin       {Begin main program}
    writeln('Which number, in the range -32767 to +32767, would you ');
    write('like to convert from base 10? ');
    readln (datum);
    for scale := 2 to maxradix do
    begin
        write('Your number in base ',scale,' is.. ');
            writenumber(datum,scale);
            writeln
         end; {for loop}
      write('Please type in any whole number and press the RETURN key to finish');
      readln(datum)
   end.     {End main program}
```

7 Traverse the parse tree in Figure 5.48 and write down the TUSIL statement from which it came.

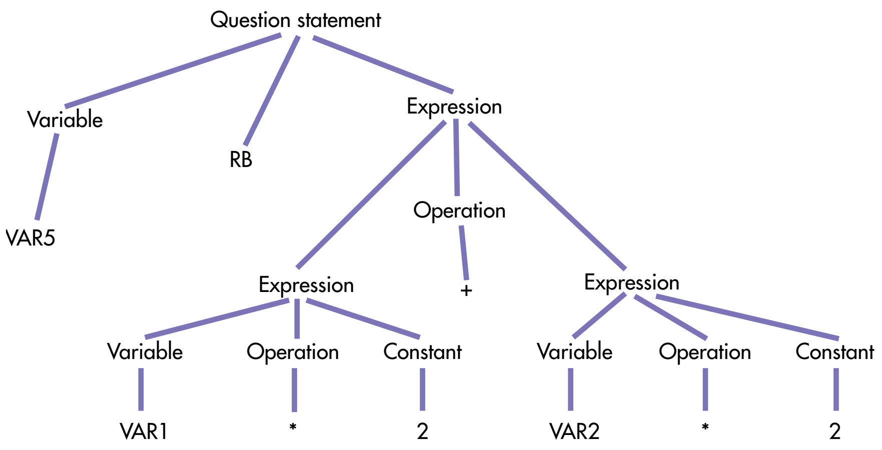


Figure 5.48

8 A spreadsheet is an example of a fourth-generation language. What type of language translator do you think a simple spreadsheet employs? Justify your answer.

Program development techniques in software solutions

As seen, there are a number of different software development approaches. However, a number of techniques can be identified in the structured approach that may also be applied to the other methods. These techniques will be looked at in terms of the structured approach, but keep in mind that they may also be applied to those other methods.

Structured approach to a complex solution

A number of different techniques can be used to aid in the development process. These techniques will make the logic of the program more visible as well as allowing a number of different people to effectively work together on the project. These techniques will also make it easier to maintain the program at a later date.

One logical task per subroutine

Each subroutine within a program should perform only one logical task. This ensures that, if the task is not performed properly, the part of the program in which the error occurred can be quickly identified. This approach also makes maintenance easier as a change in processing that involves the subroutine's task can be quickly implemented. One of the aims in creating a modular program is to allow for re-useability. Restricting each subroutine to a single task means that the module can be incorporated into another program more easily.

A further benefit of this approach to modules is its adaptability to the team approach to software development. If each subprogram is responsible for one logical task, then an individual team member can be assigned the responsibility for that module's development.

Stubs

Stubs are used to represent parts of the program that have yet to be written. Stubs can be used to test the main flow of the program or a module before all parts have been written. This technique allows a programming team to decide whether the main logic is correct. Stubs can also be used to input test data sets, such as elements of an array. This allows a programmer to test the program or its modules without the need to create the modules that retrieve and save data. For example, if a program is being written to process an array of students' marks, the test data items could be put into the array by means of a stub. This saves the developers from having to enter the marks manually or create the module to read the data from disk.

Flags

Flags are variables that indicate whether a process has been carried out or not. They are generally a Boolean variable that is initialised to 'False'. When the flagged action has been carried out, the status of the flag is changed to 'True'. It is possible to use flags that are not Boolean. For example, a flag can be used in a search routine to hold the index of the found item.

Flags are used within a program to indicate whether or not a process has been carried out. Decisions can then be made on the state of the flag. For example, if a flag is used to indicate whether an employee has worked during the week, the flag can then be tested to determine whether the wages calculation module is used or not.

Flags are also useful tools during the testing phase of program development. They can be used to determine whether a particular program section has been executed or not. If a section is suspected of causing a problem, a flag is set to 'false' before that section is to be executed. A statement that sets the flag to 'true' is placed within the section of code. Thus, if the code is executed, the flag becomes true; if it is not executed the flag will remain false. The flag can then be tested to see whether the code segment was executed.

Isolation of errors

The use of flags is one way in which errors in a program can be detected. However, there is still the problem of determining exactly where the cause of the error is. Software development systems generally include some form of help in tracing the flow of the program. In some systems we are able to manually step through the program, the contents of each of the variables being displayed in a window. Other development systems will display the line that is being executed at the time, allowing the flow of the program to be followed.

Once the cause of the problem has been identified, a remedy can be found. This may be as simple as changing the form of a condition, or it may mean a complete rewrite of the troublesome module.

As well as using these methods of detection, we may need to use one or more debugging output statements to help us to identify the cause of the error.

Debugging output statements

Temporary output statements may be placed within the code to assist with the process of debugging. These statements may be simple output statements such as one that displays the message like 'Processing of the file has been started'. Other statements may be used to display or print the values of critical variables within a module so that the values can be examined. These statements are removed from the program once the problem has been fixed.

Elegance of solution

Programmers should always be on the lookout for algorithms that are simple and clear in purpose. For example, if a flag is a Boolean variable, there is no need to code an 'IF' statement with 'IF flag is true THEN…'. A more elegant solution is to remember that the result of a condition must be either true or false. This means that the 'IF' statement can be written as 'IF flag THEN…'.

Elegant solutions to a problem are simpler algorithms and therefore easier to understand and, due to the simplicity, will form smaller and therefore faster code.

Writing for subsequent maintenance

This course has already discussed the need for coded solutions to contain internal documentation (notes or remarks) and to use intrinsic documentation (appropriate choice of identifiers). Other aspects that can help maintain a solution include the structure of the algorithm and code and the use of identifiers to represent constants as well as variables.

When we write an algorithm or code it in a language, a conscious effort should be made to create a structure that is easily read and understood. For example, subprograms should be defined together. Coded repetitions should be identifiable by the use of indentation, so that the processes that are carried out within included loops are visible. The consistent use of structures also aids in maintenance.

It is tempting to include the values of constants only in the lines of code where they appear. For example, if the rate of tax is 20%, we might write a line of code such as 'tax := pay * 0.2'. However, assigning a constant the value 0.2 at the beginning of the module has the distinct advantage that, if the value has to be changed at a later date, it has to be changed only at the beginning of the module. This eliminates the need to search through the code to try and find the values.

The process of detecting and correcting errors

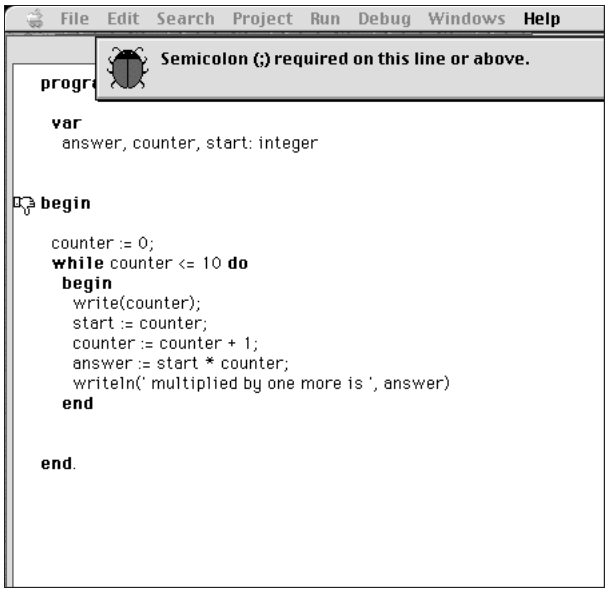
It is very unlikely that a significant program will be developed without any errors. Since these errors will either stop the program from working at all or stop it from performing all its required tasks, they must be detected and corrected. Some of the errors will be detected during the design phase, when the algorithms are checked; others will not appear until the program is translated; others will appear when the program is executed. A developer needs to develop the patience and skill necessary to detect and correct these errors. This section examines the types of error that can occur and the methods employed to correct them.

Syntax errors

Program errors will fall into one of two categories: errors in syntax or execution errors. Frequently, logical errors will have been removed by this stage by following the sequence of the program development cycle. However, some logical errors may only surface when the program is running and able to be tested using the appropriate test data.

The manner in which the structure of a language is shown was examined in the Preliminary Course and earlier in this chapter. The syntax, which governs the use of the language, is used during the translation process, as explained earlier in this chapter. During syntactic analysis the tokens are placed in a parse tree. If a token cannot be placed in position in the tree according to the rules of the language, an error message is generated. These errors are known as syntax errors. Syntax errors are identified by the programmer as written statements that do not conform to the rules of the language. This type of error is, therefore, dependent on the language used. Variations in the implementation of languages may render a legal statement on one computer platform illegal on another. Most syntax errors, however, occur through typographical mistakes when the code is being prepared on the text editor. Care with typing can assist in reducing the number of errors. A further aid in error detection, both with syntax and run-time errors, is to have one or more people desk check the program steps as if they were the computer compiling and running the program. As far as possible, the author of the program should not participate in this check.

Early compilers were very curt in providing information about the problem in the source code. Statements such as ERROR 002 IN LINE 23 or ERROR 35 gave little indication of the form of the error; this had to be determined by the programmer from the compiler documentation. Early personal computers, such as the Apple II and Commodore 64, often came with a BASIC interpreter in ROM. These interpreters, again, were very curt in their error indication, issuing the message SYNTAX ERROR IN LINE… but giving no indication as to the form of the error. The main reason for the brevity of the message was the need to create the compiler or interpreter so that it would occupy minimal memory. As computing technology has advanced and memory has become available in larger quantities, the assistance that can be provided by the interpreter has increased. Current generation compilers and interpreters assist the programmer by providing not only a message stating the error but also an indication as to where the error might occur and how it may be corrected.



Semicolon (;) required on this line or above.

```
progr
 var
  answer, counter, start: integer

begin

 counter := 0;
 while counter <= 10 do
  begin
   write(counter);
   start := counter;
   counter := counter + 1;
   answer := start * counter;
   writeln(' multiplied by one more is ', answer)
  end


end.
```

Figure 5.49 Missing punctuation is a common cause of a syntax error.

Common causes of syntax errors include:

- missing or wrong statement punctuation, for example failure to close parentheses, missing commas, missing semicolons or the wrong type of parentheses for the statement type.
- typographical errors in reserved words, for example the word REPEAT being entered as REPET
- failure to complete groupings such as following a REPEAT with UNTIL at the end of a loop
- incomplete program statements, for example starting an IF statement but omitting a condition statement such as WAGE > 40000.

Compile-time errors may be caused by factors other than a badly constructed program statement. In languages that require identifiers to be declared before use, a syntax error will be generated by the use of an undeclared identifier. The cause of an undeclared identifier may be either a typing mistake or the inadvertent omission of the identifier from the declaration section of the program.

Other syntax errors include:

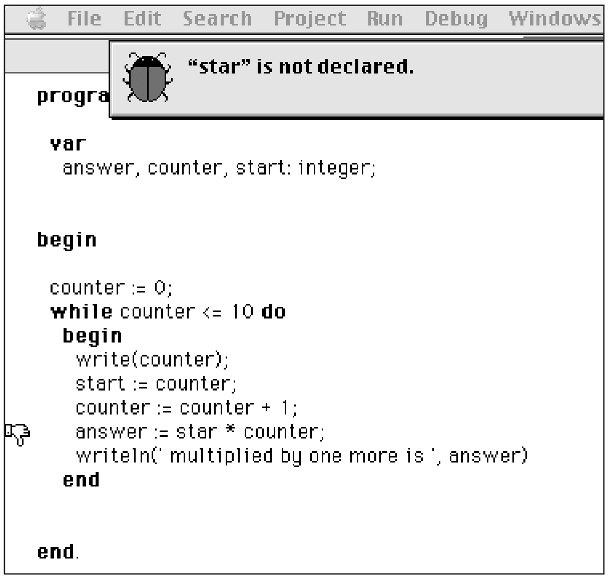
- the use of an undeclared identifier
- a mismatch in the types of variable within an assignment statement, for example trying to assign a numerical value to a variable of type character
- the use of a reserved word as a variable identifier or, in some languages, within an identifier. (For example, the compiler may reject the use in Pascal of the variable end_value as end is a reserved word.)
- The use of strings of characters over a certain length (for example in displayed messages) or too great a complexity in calculation. In both cases, splitting the text or calculation into two or more parts, each occupying its own statement, will eliminate the errors.

Logic errors

Errors in logic are usually detected at the algorithm checking stage, but it is still possible for these errors to appear after the program has been coded. A desk check of the coded module should bring these errors to light. However, logic errors can remain in a program after the desk check has been performed, if test data items are poorly chosen.

Other errors in logic may occur during the coding process, when a structure used in the algorithm description is not available in the programming language. For example, in early versions of BASIC, the multiple selection (CASEWHERE) was not available. When coding programs in this language, a number of nested IF statements had to be used. The order of the nesting affects the outcome of the program. Other causes of logic errors may come from changes that have been made to correct other errors.

Once the program has been coded, the detection of logic errors needs to be done in a logical and structured manner. First the module in which causes the error has to be found, then the cause of the error needs to be identified and corrected. A number of techniques can be used to detect the source of errors. These techniques include desk checking and peer checking of the code as well as placing breakpoints in the code and using debugging output statements. Some of these techniques have already been discussed, the others will be explained further in the next section.



"star" is not declared.

```
progra

 var
  answer, counter, start: integer;


begin

 counter := 0;
 while counter <= 10 do
  begin
   write(counter);
   start := counter;
   counter := counter + 1;
   answer := star * counter;
   writeln(' multiplied by one more is ', answer)
  end


end.
```

Figure 5.50 An undeclared identifier is one form of syntax error.

Peer check

The author of a particular work, whether it is a book, music or a computer program, is too close to the project to be able to critically evaluate it. In order to overcome this problem, programmers who are of equal status to those who wrote the code perform the checking. This process is known as **peer checking**. The purpose of peer checking is to objectively identify any errors.

Because the programmer is 'on the same level' as the checker the comments made about the operation of the module can be acknowledged without having to consider the relative position of the reviewer.

Desk checking

As already seen, desk checking is the process whereby an algorithm is manually worked through with the test data items. The results of operations on variables are tabulated on paper together with the outputs that occur. By comparing these results with the expected outputs, the algorithm can be checked. A similar desk check should also be performed on the coded module before it is translated. In this way a number of errors can be eliminated before the computer is used.

Use of expected output

When test data items are created, the expected output of processing is included. This allows the programmer to determine whether the algorithm performs its task correctly. The outputs should be tested both after the algorithm has been designed and when the program has been coded. If the outputs from the checking process match, then, provided the test data has been carefully selected to test all parts of the module, the module should work correctly or all the errors in the module should have been identified.

Run-time errors

Once a program has been successfully compiled, it is still untested. It now has to be executed with the test data. This phase is crucial to the success of the final solution. Each of the test data elements is used, the resulting output being matched to the expected values. Modifications to the code may be required at this stage if the program does not perform as expected.

Other errors that affect the running of the program will also become evident at this stage of development. These run-time errors will be treated in more detail in the next section of this chapter.

Once tested with the test data, the program should be given to other users to try to cause the program to fail or crash. The first version of a program is often referred to as the alpha version and a more stable version, which is closer to the final product, is known as the beta version. Independent operators, whose task is to uncover as many execution errors as possible, often perform the alpha testing and beta testing.

Program testing at this stage may also involve testing with different operating systems, different versions of operating systems and different hardware combinations in order to provide information for the user documentation and installation guides.

Syntax errors are relatively easy to locate and correct, but run-time errors can be difficult to locate. A run-time error in a program may not be evident at the point at which it occurs, but may surface later, causing all kinds of trouble in the program.

The first kind of error involves one or more data values that cause the computer to attempt a calculation for which there is no result or for which the result is not as anticipated. For example, a division by zero is unable to be evaluated and so the operating system will cause an error to be generated. This type of error will be signalled at the point in the program where it occurs, usually halting execution, so it is easy to locate and correct. Other errors falling into this category are those in which the tangent ratio of 90° is to be used and some results of integer arithmetic.

Example

Integer arithmetic can cause problems if the result of an operation is outside the range allowed for that machine (typically between –32768 and 32767). If, for example, 32766 and 32760 are added using integer arithmetic, a result of –10 occurs, which is clearly not the correct result. This problem may be overcome by reordering the operations in the statement or choosing a different data type for the variable(s).

Calculations performed by a computer using real numbers are subject to errors caused by the inability of the computer to exactly store decimal values. A small part of the value may be 'lost'. This process is called 'truncation'. In many cases these errors are insignificant and will not cause problems for the programmer; however, if there are a large number of calculations to perform, the order of operations may make a difference. In this case a reordering of the calculations may improve the accuracy.

A further problem, caused by the manner in which real numbers are stored, is that of equality. If a decision is to be made on whether a value is equal to another or not, a wrong result may be obtained if the tested value is close. The reason for this is again the manner in which calculation results are stored in main memory.

Example

The following BASIC program illustrates this problem:

```
10  INPUT NUMBER
20  ANSWER = NUMBER / 9999999999
30  ANSWER = ANSWER * 9999999999
40  IF ANSWER = NUMBER THEN PRINT 'equal' ELSE PRINT 'not equal'
50  END
```

The above program does not perform the expected match as the stored value of ANSWER is truncated in line 20. When the multiplication in line 30 takes place, a slightly different value is obtained, so the message 'not equal' is displayed. By rewriting the program in the following manner, a correct result is obtained each time the program is run:

```
10 INPUT NUMBER
20  ANSWER = NUMBER / 9999999999 * 9999999999
40  IF ANSWER = NUMBER THEN PRINT 'equal' ELSE PRINT 'not equal'
50 END
```

A second type of error involves control not following the paths as designed in the algorithm description. This kind of error is much harder to rectify and involves using techniques such as setting breakpoints and tracing the flow of control. Breakpoints are places in the program where execution is temporarily suspended so that the programmer can determine whether the program flow reaches that point. A breakpoint is often placed after a printout of values of variables. This allows the programmer to examine the values before resuming the program. 'Tracing' refers to a display on the screen of the path taken during execution of a program. Some languages support a trace function (for example many versions of the BASIC language), but statements can be placed in the code of a subprogram to achieve the same result. These statements may be simple messages such as 'This is the barcode search module'. Other aids that may be incorporated into a program development system include the ability to step through a program one instruction at a time and the ability to display the values of each of the variables at each stage of execution.

A further problem may be caused by the program trying to access inappropriate memory locations. For example, a loop may pass through the elements of an array in sequence. If the loop has not been properly terminated, it may try to access an array element that does not exist. This will cause a problem. For example, an array has been indexed from 1 to 20. If during the execution of a loop the counter is incremented, the array element is accessed before the termination test has been made and it is possible for the array element number 21 to be requested. In this case, an inappropriate memory location has been requested and will thus cause an error. This type of error will usually result in a run-time error message. However, if a first- or second-generation language is being used and an inappropriate memory location is accessed, the program will continue causing an error in output further along in the processing.

Once the run-time errors have been located, they need to be corrected. The process of correction may be as simple as changing a line of code or adding a new line, or it may require a rewriting of an algorithm and coding a completely new module.

The use of software debugging tools

Many modern software development environments do not consist of just a text editor and a translation system. They are often integrated with a number of tools that have been designed to help the programmer with the process of error detection and correction. One of these tools that we have already met is the syntax error message that gives an indication of the probable cause of the problem, where that cause is and a suggested solution to the problem. Other tools can be used to assist in determining the problems that occur within a running program.

Use of breakpoints

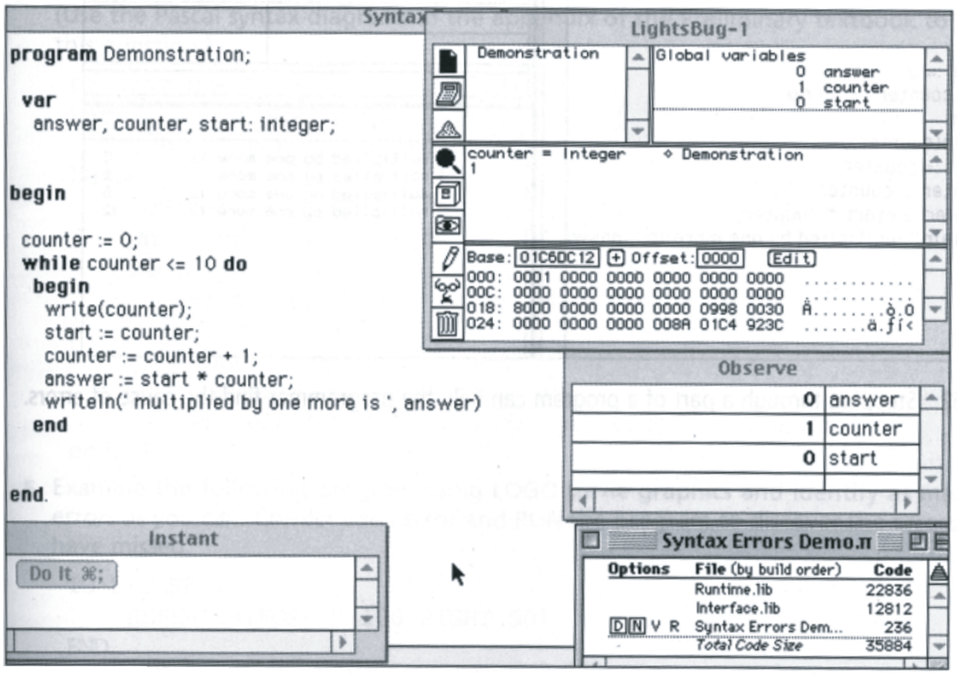
Breakpoints are places where the program is made to stop during execution. After the break in execution, the program can be made to resume operation or halted. Breakpoints are usually placed in the code by the programmer inserting a 'break' command within the source code. Some programming environments may include ways of creating breakpoints at critical points within the code.

The main purpose of a breakpoint is to allow the programmer to examine the contents of memory locations during program execution. Some programming environments allow the programmer to list the variables that are to be examined in a separate window. If the variable values are left unchanged at the break, execution of the program may be resumed. Some programming environments allow for the variables to be changed at a break, with execution able to be continued after the change. Other environments will not allow the resumption of execution after a break.

By the careful use of breakpoints, it is possible for a programmer to quickly home in on the source of a problem in the code.

Resetting variable contents

The values of variables can also be reset within a running program by either taking advantage of breakpoints or placing statements within the code that change the values of those variables to known values. In this way the effect of processing within one section of code on known data values can be seen. When the section of code has been debugged, these extra lines can be removed. This can be especially useful when we have data that is supposed to have been read in from a file. By using this technique, we can be sure that the data values being processed are exactly those for which we know the outputs.



```
program Demonstration;

 var
  answer, counter, start: integer;

begin

 counter := 0;
 while counter <= 10 do
  begin
   write(counter);
   start := counter;
   counter := counter + 1;
   answer := start * counter;
   writeln(' multiplied by one more is ', answer)
  end

end.
```

```
Demonstration          Global variables
                              0   answer
                              1   counter
                              0   start

counter = Integer      ◇ Demonstration
1

Base: 01C6DC12  + Offset: 0000   Edit
000:  0001 0000 0000 0000 0000 0000   ............
00C:  0000 0000 0000 0000 0000 0000   ............
018:  8000 0000 0000 0000 0998 0030   Ä.........ò.0
024:  0000 0000 0000 008A 01C4 923C   .......ä.ƒí<
```

```
Observe
      0  answer
      1  counter
      0  start
```

```
Syntax Errors Demo.π
Options    File (by build order)       Code
           Runtime.lib                22836
           Interface.lib              12812
D N V R    Syntax Errors Dem...         236
           Total Code Size            35884
```

Figure 5.51 THINK Pascal provides a number of different methods to detect errors.

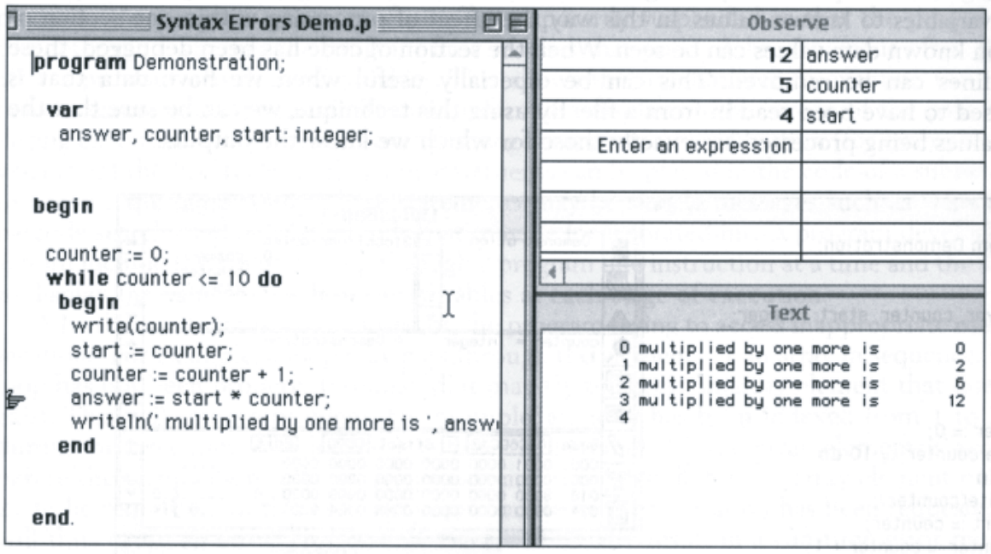
Program traces

A programmer can learn a great deal about the working of a program if the order in which the statements are executed is known. In many software development systems a programmer can follow the order of execution by printing or displaying the line number or identification of each statement as it is executed. This process is known as **tracing**.

Tracing provides the programmer with a means of comparison. The actual flow of execution can be compared with the expected flow from the source code and from the original algorithm. Armed with this information, the programmer can often pinpoint the cause of problems within the module.

Single line stepping

When trying to find the cause of a problem, a programmer will often need to follow the values of variables from line to line as the program is being executed. Many program development systems allow the programmer to 'step' the program. In this process the program is executed one line at a time. After the line has been executed, the computer halts, waiting for the programmer to tell it to execute the next line. It is also quite common for this process to be accompanied by an indication of the line that is being processed together with a display of programmer-chosen variable values. A programmer can use this technique to identify the line or lines of code responsible for a problem. This technique is usually only used when a small section of code has been identified as containing an error. It would be physically impossible for a programmer to step through a very large program line by line. When a program development system allows for stepping, it will also allow a programmer to jump through sections of code such as loops.



```
program Demonstration;

  var
    answer, counter, start: integer;


begin

  counter := 0;
  while counter <= 10 do
   begin
     write(counter);
     start := counter;
     counter := counter + 1;
     answer := start * counter;
     writeln(' multiplied by one more is ', answw
   end


end.
```

12	answer
5	counter
4	start
Enter an expression	

```
0 multiplied by one more is      0
1 multiplied by one more is      2
2 multiplied by one more is      6
3 multiplied by one more is     12
4
```

Figure 5.52 Stepping through a part of a program can help the programmer find the cause of errors.

Exercise 5.5

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Each __________ in a program should perform only one __________ task. This makes the ______ of the program easier to follow in the __________ and __________ stages of the development cycle. Another aid in the testing __________ is the use of __________ to represent parts of the code that have not yet been __________. We can also use __________, debugging __________ statements and __________ the program flow to help locate the source of an error.

2 Explain the terms 'syntax error' and 'run-time error' in your own words. Give examples of syntax errors to illustrate your answer.

3 There is an error in each of the following segments of code. Identify the most likely error and correct it.

a `COST PRICE—DISCOUNT`

b `(SETQ (PLUS (A B))`

(This is a small LISP program)

c
```
while counter < 10 do begin
time := 0
repeat
   time := time + 1;
   write(time);
end;
```

(Use the Pascal syntax diagrams in the appendix of the Preliminary textbook to help with the following questions.)

d `write('This message will appear on the screen to help you);`

e `if then index:= index + 1`

4 Examine the following program in Pascal and identify as many syntax errors as you can. Correct each error and compile the program to discover the errors you may have missed. (Use the Pascal syntax diagrams in the appendix of the Preliminary textbook to help with this question.)

```
program question_3
var
    answer, counter : integer;
begin
    counter := 0;
    while counter <= 10 do
    begin
        write(counter);
        start := counter;
        counter := counter + 1
        answer := start * counter;
        writeln(' multiplied by one more is , answer)
end.
```

5 Examine the following program using LOGO turtle graphics and identify as many syntax errors as you can. Correct each error and RUN the program to discover the errors you may have missed.

```
TO SQUARE
    REPEAT 4[FORARD 100 RIGHT 90]
END
```

```
TO CIRCLE
    REPEAT 360[FORWARD 1 RIGHT 1
TO HOUSE
    SQUARE
    TRIANGLE
END
TO TREE
    FORWARD 100
    RIGHT 90
    CRCLE
    LEFT 90
    BACKWARD 100
END
```

6 Examine the following BASIC code and identify as many syntax errors as you can. Correct each error and run the code to discover the errors you may have missed. (You may have to modify some of the code to run on your system.)

```
10   COUNT = 0
20   WHILE COUNT < =
30       PRINT COUNT;
40       START COUNT
50       COUNT = COUNT + 1
60       ANSWER = COUNT START
70       PRINT ' multiplied by one more is ;ANSWER
80   WEND
90   END
```

7 Examine the following code in Hypertalk and identify as many syntax errors as you can. Correct each error and execute the code to discover the errors you may have missed. You will need to create two card fields called Fahrenheit and Celsius.

```
on mouseUp
     st numberFormat to 00.00
     get card 'Fahrenheit'
     subtract 32 from it
     multiply it by 5
     divide it by 9
     put it into card field 'Celsius'
end
```

8 Describe in your own words the meaning of the term 'run-time error'. Give examples to illustrate your answer.

9 Describe the tools that can be used to help the software developer locate errors in a coded program. Explain how these tools may be used to locate the source of a run time error in a program.

10 Choose a public domain or shareware program and test it, documenting any problems you find.

11 Create a small program that asks for the entry of two numbers and outputs their product. Write the code in two different ways, the first way treating both numbers as integers, the second way treating the numbers as floating point values. What effects do each of these methods have on the stored result?

12 Explain how you could trace the path through a program while it is being executed. What purpose would a trace serve in the debugging process?

13 Explain why the process of desk checking may not detect errors. Illustrate your answer with examples.

Documentation of a software solution

Software documentation performs three tasks: it tells what the software is to do, how it is to do it and what a user needs to do in order to run the program. Documents such as the program specifications, algorithm description and code listing are used to describe the aims of the program and its methods of solution of the problem. These documents are useful for the programmer in both the development and maintenance stages of the programming development cycle. User manuals, installation guides, tutorials and online help are provided for the operator.

Documentation should be created during each stage in the development cycle and not left as an afterthought to be done at the end of the process.

Forms of documentation

During the system development cycle a large amount of documentation is produced. Some of these documents will form the basis of the product documentation, while the rest, known as the process documentation, will become largely outdated. Product documents fall into two categories: those that become the system manuals and those that form the user's guides. System documentation is provided to assist with the maintenance of the system and will contain many of the documents produced during the system development cycle. User documentation incorporates documents that describe the purposes of the system and how the end user can use the software. A second type of user document is provided for system administrators whose task it is to keep the system running.

Process documentation is documentation produced as a by-product of the system development cycle, and which will have served its purpose by the time the cycle reaches the implementation stage. Documents falling into this category include test schedules, memos, working papers and reports.

Product documentation produced at various stages during the system development cycle includes the program specifications, dataflow diagrams, data dictionary, output specifications report and data files specifications.

As can be seen from the above descriptions, final documentation of the product is not created as an afterthought but forms an integral part of each stage of the development cycle. It is extremely important that the documentation is kept current, by being updated each time changes are made.

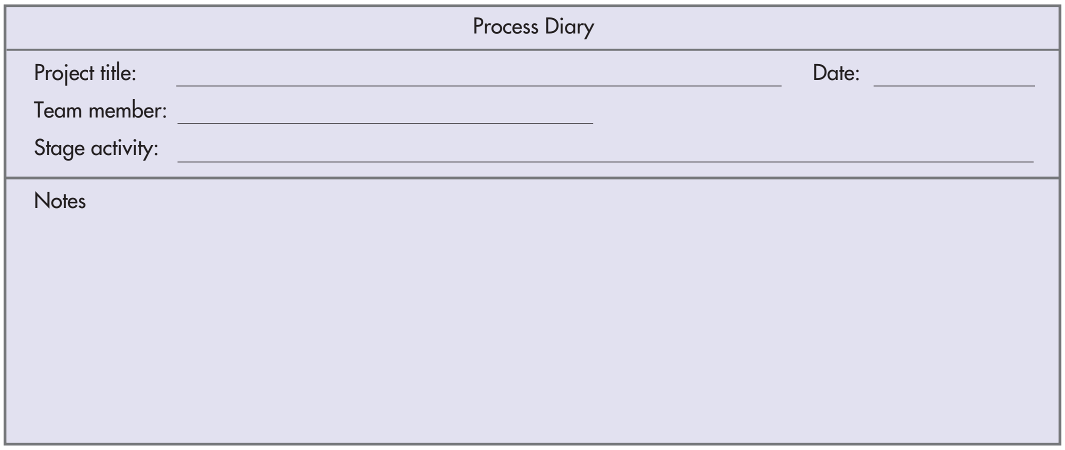
The design process itself is an important aspect of documentation. The original design plan, together with subsequent modifications, can be useful in planning further design activities. Such a design history can be extremely useful in later projects. Design processes learnt during the development of one system can lead to greater success when new projects are attempted.

The purpose of product documentation is twofold: to provide a detailed description of the system and to provide information that will assist with the maintenance of the system. System documentation consists of a number of documents produced during the development cycle: the system requirements, system descriptions, algorithm descriptions, program source code and a system maintenance guide.

Process diary

A detailed record of the process that has been followed during the development of a software project is an important tool for a developer. The information provided by the diary is useful in a number of ways. First, it allows a new member of the team to quickly become familiar with the processes that have already taken place. A process diary also documents the milestones and pitfalls that have been met along the way. This record can be an important reference when a similar project is undertaken in the future. Maintenance programmers can use the diaries to understand the processes that were originally undertaken. Armed with this information, they can avoid problems met by the original developers.

The actual layout of a process diary varies greatly, but they do need some common features. The diary should at least contain some reference to the time of the occurrence. This reference could be as broad as a reference to a week or narrowed down to a particular time. This will allow the diary's reader to follow the sequence of events. Descriptions of the events that took place within the development process are also needed. Other aspects that may be covered by the diary, especially if all members of a team are using a single diary, are details such as the team member making the note and the team description.



Process Diary

Project title: ______________________ Date: __________

Team member: ______________________

Stage activity: ______________________

Notes

Figure 5.53 A process diary may take many forms. Here is one example.

User documentation

Documentation required by the end user will consist of a number of documents: a functional description, an introductory manual, a system reference manual, a system installation guide and a system administrator's manual. In addition, support documents such as reference cards and on-line help are often provided to assist the user.

A **functional description** contains a brief outline of the system requirements and the purpose of the system.

An **installation guide** may be the first contact that a user has with the program documentation, and the programmer should bear this in mind when designing the installation documents. The guide needs to specify the minimum hardware and operating system requirements as well as the manner in which the software is to be installed. The guide must be clearly written in simple, non-technical language and should describe all steps taken to install the software. It should also be written in a manner that is neither condescending nor overly technical. Illustrating the process by means of screen dumps or screen instructions will further assist the user to successfully load the application on to the system. A description of problems that may be encountered should be included in the guide.

Special considerations such as the deactivation of virus detection software should be identified in the installation manual and made as simple as possible for the user. By creating an installation guide that is accurate, friendly and clear, the programmer will give the user confidence in using the application. It does not matter how good a program is, if difficulty is encountered in the installation stage the user immediately loses faith in the product.

As with all other forms of documentation, the installation guide needs to be updated if there are any changes made during system maintenance.

An **introductory manual** describes the manner in which the system is started and the use of common system functions. The introductory manual should contain a series of tutorials that illustrate these common functions as well as methods of recovering from common errors. It is important that the introductory manual is written in an informal and non-technical manner so that a beginner will find it easy to follow.

A more comprehensive **system reference manual** will accompany the introductory manual. The system reference manual will contain a complete listing of all the functions of the system, arranged in a logical manner (for example listing the functions alphabetically or by family). Each of the functions should be described in detail, with a description of the inputs it requires, the outputs it produces and any special features it has. As well as these basic descriptions, the manual may contain samples of screen displays that the user might encounter while using the functions, and a graphical representation of the way in which the functions relate to each other. A second purpose of the manual is to provide a complete description of all known operational errors and how to recover from them.

The system reference manual should be written in a more formal style than the introductory manual and provide a comprehensive description, as its purpose is for reference. Readability is not as important as accuracy and clarity. Language used in this manual will tend to be more technical than in the introductory manual, although the use of technical terms should be kept to a minimum.

A **system administrator's manual** is designed to provide the system administrator with a comprehensive description of the program's interaction with other elements of the system and other systems to which it is connected. It will document all messages created when the system interacts with other systems and how to respond to those messages. Language used in this manual will tend to be more technical than in the other documentation; however, technical terms should not be used unless absolutely necessary.

A **user manual** is the main point of reference for a program's user. It needs to be clearly and logically set out, and needs to contain both technical and tutorial information. It should be fully indexed so that the user can access any item quickly. The manual should contain a description of each of the program's functions, examples of the function's uses and the methods of navigation to each function.

The language used in the user manual, as with the installation manual, should be clear, concise, non-threatening and non-condescending. The purpose of the manual is to instruct the user as to how a particular action may be undertaken and to assist the user in becoming more confident in using the program.

The layout of the manual depends on the program, but the following are the most popular presentations:

- The information in the manual is arranged in the order in which a user is most likely to encounter or need it. This type of manual will often be written as both a technical manual and a tutorial.
- The items are arranged in the order in which they appear in a menu. The major menu headings provide the chapter breaks and the menu options are described within those chapters.
- The items are ordered alphabetically. This approach is useful if there is a large number of items to be covered. A disadvantage of this system is that, unlike the previous two methods, the relationship between the elements is not immediately apparent and may take an amount of cross-referencing to sort out.
- An option that is gaining popularity is to provide the manual as some form of online help. The advantages of this method are that the manual is available to the user at the time it is required, and the computer's ability to search and store can be used to find information as well as to keep track of the search paths followed.

The user manual should contain a series of tutorials that cover all aspects of the program's use. These tutorials may form part of the user reference or they may be issued as a separate document. Regardless of the method of issue, the tutorials should be supplemented by a number of prepared files that can be used to produce the desired results. The prepared files can contain some of the test data where appropriate. By using these tutorial files, screen displays can be captured and presented to the user in order to confirm that the approaches taken are, indeed, correct. A good tutorial can do much to improve the user's confidence with both the software and the hardware.

Example

ClarisImpact is a business graphics package that offers the user a way of producing multi-page drawings, reports and presentations. The Getting Started manual (Figure 5.54) is divided into six chapters. The first is an installation guide, the second provides a general overview of the application, and the other four deal with the major features of the program. Each of the operational chapters uses a series of tutorials to take a new user through the basic functions of the program. At the end of each chapter is a reference table that directs the user to the appropriate chapter in the User's Guide (Figure 5.55) where more information can be obtained.

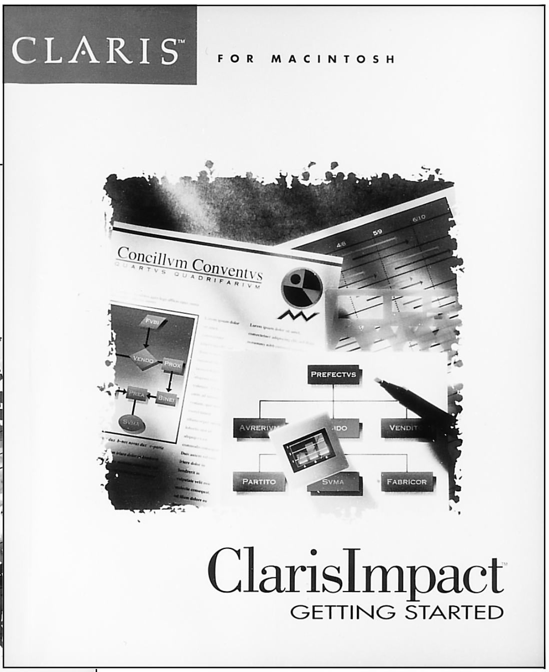


Figure 5.54 The ClarisImpact Getting Started manual allows the user to have the program installed and running quickly.

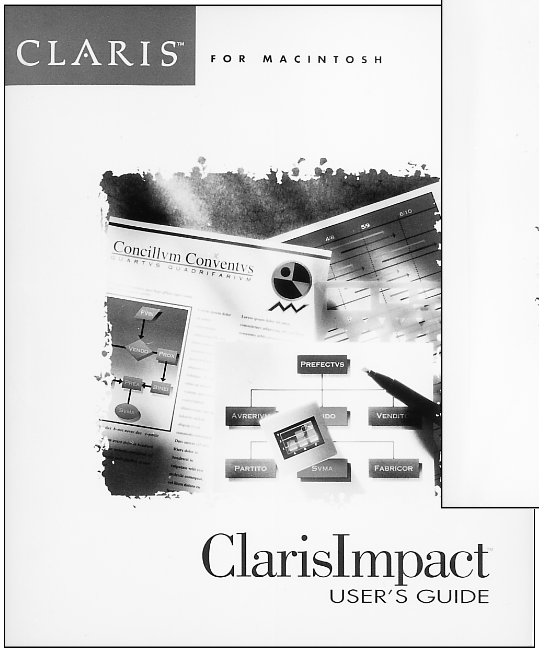
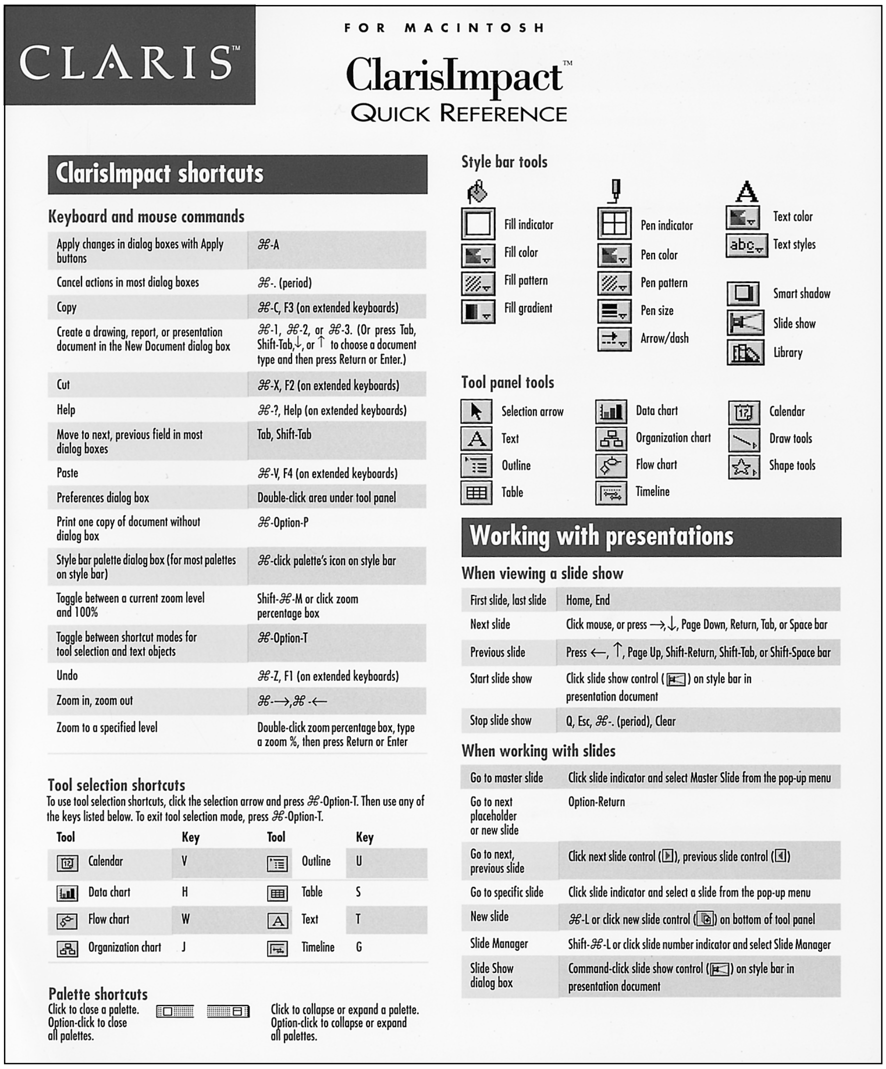


Figure 5.55 The ClarisImpact User's Guide is a more detailed document that explains the working of the program.

Reference cards, which contain a brief list of common system functions and how to use them, can provide support to both experienced and first-time users. A reference card is usually one sheet containing a minimal description of common functions. It is designed for ease of use, its purpose being to provide information without reference to the main set of manuals. Colour coding of functions and key combinations will provide the user with visual clues to help in the use of the card.

The Quick Reference card supplied with ClarisImpact (see Figure 5.56) provides a summary of the program's shortcuts and functions available in each of the modules. The card has a picture of each of the icons used in the program, with the name of the function it represents beside it.

Online help is becoming a more common method of providing the user with assistance. This form of documentation has a great advantage over paper-based manuals in that it is



CLARIS™

FOR MACINTOSH

ClarisImpact™
QUICK REFERENCE

ClarisImpact shortcuts

Keyboard and mouse commands

Apply changes in dialog boxes with Apply buttons	⌘-A
Cancel actions in most dialog boxes	⌘-. (period)
Copy	⌘-C, F3 (on extended keyboards)
Create a drawing, report, or presentation document in the New Document dialog box	⌘-1, ⌘-2, or ⌘-3. (Or press Tab, Shift-Tab,↓, or ↑ to choose a document type and then press Return or Enter.)
Cut	⌘-X, F2 (on extended keyboards)
Help	⌘-?, Help (on extended keyboards)
Move to next, previous field in most dialog boxes	Tab, Shift-Tab
Paste	⌘-V, F4 (on extended keyboards)
Preferences dialog box	Double-click area under tool panel
Print one copy of document without dialog box	⌘-Option-P
Style bar palette dialog box (for most palettes on style bar)	⌘-click palette's icon on style bar
Toggle between a current zoom level and 100%	Shift-⌘-M or click zoom percentage box
Toggle between shortcut modes for tool selection and text objects	⌘-Option-T
Undo	⌘-Z, F1 (on extended keyboards)
Zoom in, zoom out	⌘-→,⌘ -←
Zoom to a specified level	Double-click zoom percentage box, type a zoom %, then press Return or Enter

Tool selection shortcuts

To use tool selection shortcuts, click the selection arrow and press ⌘-Option-T. Then use any of the keys listed below. To exit tool selection mode, press ⌘-Option-T.

Tool	Key	Tool	Key
Calendar	V	Outline	U
Data chart	H	Table	S
Flow chart	W	Text	T
Organization chart	J	Timeline	G

Palette shortcuts

Click to close a palette. Option-click to close all palettes.

Click to collapse or expand a palette. Option-click to collapse or expand all palettes.

Style bar tools

Fill indicator	Pen indicator	Text color
Fill color	Pen color	Text styles
Fill pattern	Pen pattern	Smart shadow
Fill gradient	Pen size	Slide show
	Arrow/dash	Library

Tool panel tools

Selection arrow	Data chart	Calendar
Text	Organization chart	Draw tools
Outline	Flow chart	Shape tools
Table	Timeline	

Working with presentations

When viewing a slide show

First slide, last slide	Home, End
Next slide	Click mouse, or press →,↓, Page Down, Return, Tab, or Space bar
Previous slide	Press ←, ↑, Page Up, Shift-Return, Shift-Tab, or Shift-Space bar
Start slide show	Click slide show control ( ) on style bar in presentation document
Stop slide show	Q, Esc, ⌘-. (period), Clear

When working with slides

Go to master slide	Click slide indicator and select Master Slide from the pop-up menu
Go to next placeholder or new slide	Option-Return
Go to next, previous slide	Click next slide control ( ), previous slide control ( )
Go to specific slide	Click slide indicator and select a slide from the pop-up menu
New slide	⌘-L or click new slide control ( ) on bottom of tool panel
Slide Manager	Shift-⌘-L or click slide number indicator and select Slide Manager
Slide Show dialog box	Command-click slide show control ( ) on style bar in presentation document

available to the user without leaving the computer. This is especially important for network users who may be in different locations. Online help can also be linked to the menu items, for example providing a description of the function of a button by means of a pointer.

Two common forms of online help are 'balloons', which contain a description of the menu or screen item that is pointed to, and help screens, which describe each function on-screen. A particular benefit of the help screen is that it can make use of a computer's ability to rapidly search through a database of items for a wanted function.

ClarisImpact provides two forms of online help: balloons and a topic-based help program. Balloon help gives the user information about the various screen elements. The topic-based help program describes the steps taken to perform various tasks. Both forms of assistance are available from the menu bar.

Murphy's Law of 'What can go wrong usually will' applies to all software. A computer solution to a problem has so many different variables that there will be times when all does not go well. It is at this time that the user turns to the **trouble-shooting guide** for reassurance and guidance.

A good trouble-shooting reference will cover all the common problems that may occur, whether it is the fault of the software or some other system element. The reference should also, as far as possible, deal with the more obscure problems, such as possible conflicts with operating system modules.

Error messages on a screen, by their nature, must be brief and to the point. In most cases it is not possible for the screen message to pinpoint the exact cause of a problem, and this is where the trouble-shooting guide takes on its major role. The layout of the manual should provide for easy location of the solution to the problem. Thus the guide should contain a thorough index that is cross-referenced to both the nature of the problem and the screen message. It should deal with similar problems in the same sections of the manual and should provide a clear explanation as to how the problem can be overcome. Sample screens and messages should be used to assist the user to identify the problem and illustrate its solution.

The language used within the guide should be chosen with care, as the user can lose confidence when there is an error. As with the other user documents, the use of condescending, excessively technical or humorous language should be avoided. What the users need at this stage is assistance in solving the problem and reassurance about their ability to use the hardware and software.

A reduction in the cost and an increase in the power of computer hardware have given programmers the opportunity to use the computer to provide some of the documentation required. Several methods of providing online documentation have been devised; the most common used are balloon text, user instructions and tutorial links.

Balloon text is usually associated with a graphical user environment such as Windows and the Macintosh operating system. When enabled, balloon text will describe the action of a display element whenever the screen pointer points to that element.

The placement of a set of **user instructions** on the screen is the easiest of the screen aids to implement, as it requires only the addition of a line or two of output text. However, it is often difficult to phrase the instructions so that they are clear to the user. The instructions should be placed in such a position that they leave a majority of the working screen for the application display. The most common positioning of user instructions is at the bottom of the screen; this is especially true in a text-based display.

A documentation method that is gaining popularity is the use of a **tutorial-type assistant** that helps the user complete a task. This kind of help presents options to the user at various stages of the process, assisting with the steps. These online assistants have been designed to provide the user with the necessary information to complete a task, often offering the most common choice as a default.

Self-documentation of the code

The program listing is probably the most obvious documentation associated with a computer-based solution to a problem. However, the manner of presentation of a program can affect both its legibility and its ability to be followed and/or modified. For example, compare the legibility of the following two samples of identical Pascal code. The first sample has not been formatted to any standards; the second has been coded with one instruction per line and indentation has been used to show the statement levels.

Sample 1

```
program menu_test (input, output); var choice: char;
procedure Add_record; begin writeln('Add record stub') end;
procedure Delete_record; begin writeln('Delete record stub') end;
procedure Sort_records; begin writeln('Sort records stub') end;
procedure Invalid_input; begin writeln('Invalid choice stub') end;
begin writeln('Please press the letter key corresponding to your choice');
writeln('A to ADD a record'); writeln('D to DELETE a record');
writeln('S to SORT the records'); writeln('E to END this session');
readln(choice); while (choice <> 'E') and (choice <> 'e')
```

```
do begin if choice in ['A', 'a', 'D', 'd', 'S', 's'] then
{this statement ensures that other characters are excluded … otherwise}
case choice of 'A', 'a': Add_record; 'D', 'd': Delete_record;
'S', 's': Sort_records end {end of case statement} else Invalid_input;
writeln('Please press the letter key corresponding to your choice');
writeln('A to ADD a record'); writeln('D to DELETE a record');
writeln('S to SORT the records'); writeln('E to END this session');
readln(choice); end end.
```

Sample 2

```
PROGRAM menu_test (input , output);
VAR
    choice : char;
PROCEDURE Add_record;
    BEGIN
        writeln('Add record stub')
    END;
PROCEDURE Delete_record;
    BEGIN
        writeln('Delete record stub')
    END;
PROCEDURE Sort_records;
    BEGIN
        writeln('Sort records stub')
    END;
PROCEDURE Invalid_input;
    BEGIN
        writeln('Invalid choice stub')
    END;
BEGIN
    writeln ('Please press the letter key corresponding to your choice');
    writeln ('A to ADD a record');
    writeln ('D to DELETE a record');
    writeln ('S to SORT the records');
    writeln ('E to END this session');
    readln (choice);
    WHILE (choice <> 'E') AND (choice <> 'e') DO
    BEGIN
    IF choice IN ['A','a','D','d','S','s'] {this statement
      ensures that other characters are excluded … otherwise}
    THEN CASE choice OF
                'A','a' : Add_record;
                'D','d' : Delete_record;
                'S','s' : Sort_records
            END             {end of case statement}
        ELSE Invalid_input;
    writeln ('Please press the letter key corresponding to your choice');
    writeln ('A to ADD a record');
    writeln ('D to DELETE a record');
    writeln ('S to SORT the records');
    writeln ('E to END this session');
    readln (choice);
END                             {end of while statement}
END.
```

The second sample program is easier to follow not only because of the placement of a single instruction as separate entries and the use of indentation to show the logical blocks such as loops and decisions, but because the use of upper-case letters for the Pascal reserved words together with the separation, wherever possible, of the comments allows the structure of the program to be immediately visible. Thus any modification of the code can be accomplished efficiently.

Technical documentation, including source code, algorithms, data dictionary and systems documentation

The creation of an algorithm is an interpretation of the problem specifications as a series of actions. The method used to describe the algorithm has to be clear and concise. Documentation created during this phase will be used by the coding programmer to implement the solution. (The coding programmer is not necessarily the algorithm designer as the systems analyst may provide the initial algorithms.) Algorithm documentation produced by the programming team will also be required during any maintenance stages after program implementation. The most common algorithm descriptions used at present are the two HSC-approved methods — pseudocode and flowcharting — together with structured English and Nassi-Schneiderman diagrams.

As seen, it is important that the source code contains as much information as possible about the processes being carried out. When maintenance programmers are employed to modify the program, the task is made much easier with this information. The source code can also be incorporated into a library of code for reuse. The use of libraries can reduce the development time for later projects.

Data dictionaries are also a natural product of the program development cycle. A data dictionary lists all data identifiers and their type and attributes. The attributes of a data item refer to features such as the way in which the item is displayed, the maximum size of an individual element, the range of values and default values (if any). The purpose of a data dictionary is to clearly specify the nature of all data items present in the processing stage. A further benefit of the data dictionary is that it allows the programmer to calculate the amounts of data storage required for the application. There are many different methods of presentation of a data dictionary. Some will present sample data items to further clarify the form taken by each of the different data elements. Some may also contain reference to data verification procedures.

The data dictionary is also a very useful tool for a maintenance team. It gives the team all the information required about how the data is stored and the structures that have been used. This will assist them in the task of modifying the program.

A number of other systems documents are produced during the development process. These include the requirements definition, an analysis of the working system, the original software specifications, screen and report designs and any other documents covering special features such as security of data and performance. The requirements definition sets down the needs of the user. The systems analysis report details the workings of the system and often incorporates diagrams to assist developers with an understanding of the workings of

Field	Type	Size	Range	Example
Barcode	String	13 characters	0000000000000 to 9999999999999	4891576813271
Product	String	Up to 25 alphabetic characters and spaces	a to ZZZZZZZZZZZ	Reimas Butter
Price	Floating point number	Up to 10 digits showing two decimal places	0.00 to 99999999.99	0.95

Figure 5.57 Data dictionary example.

the system. The software specifications are very important as they set out the criteria that the software needs to meet. These criteria are used in the design process as a guide in the testing stage to ensure that the software produces the appropriate outputs, and in the evaluation stage where they are used as a measure of the success of the product.

Documentation for subsequent maintenance of the code

The documentation mentioned in the previous sections is needed for product maintenance. The maintenance team has to have a clear understanding of the manner in which the software is designed and operates in order to successfully modify it.

Documents such as the algorithms and source code expose the workings of the program and provide the maintenance team with a backbone on which to build the modifications. These documents also give the maintainers a clear description of the processes that take place within the program.

Screen and report designs are used to provide the maintenance team with a basis for designing new screens that are consistent with the current ones.

Process diaries and logs provide the maintainers with an insight into the development process. Of major interest to the maintenance team are the pitfalls that occurred along the development path. This knowledge allows them to avoid similar problems when modifying the program.

Use of application software to assist in the documentation process

The documentation that accompanies a program forms the basis of human interaction with the system. There needs to be as much design effort expended on program documentation as has been spent on program development. A well-designed set of documents will have more appeal and appear more useful than a set that has been created 'because it is necessary'. Documents should invite the reader to use them.

Documentation should follow a set of simple rules for structure and presentation.

- All documents should have a cover page that identifies the project, the document, the date of production, its author and document type.
- A document should be broken up into chapters, unless it is brief, so that it can be updated easily. It is easier to replace a chapter than a whole document once it has been revised. Pages are often numbered according to the chapter in which they occur to assist with this type of maintenance (for example 2-11, B-13).
- Detailed documents should have an index, especially those that are reference documents, for example user's guides and administrator's manuals.
- Glossaries of technical terms will aid the reader to understand descriptions provided in the documentation.
- Documents should, if possible, be uniform in their presentation. Features such as page layout, page numbering, placement of indexes and glossaries and formatting should be used in a consistent manner.

Software applications such as word processors, graphics programs and databases can assist the software developer by enabling a library of templates to be created and kept. This will ensure that there is a high degree of consistency between different documents.

Word processors can be used for a number of tasks within the software development cycle. These will range from the creation of reports through to the development of user documents. Simple flat file database management systems can be used to develop data dictionaries and for process diaries. Project management software can be used to manage the whole of the development process, including the allocation of resources and time. Graphic systems can be used to prepare screens and desktop publishing systems to create report designs. Almost any productivity software can be used at some place within the software development process.

Use of CASE tools

CASE tools provide a purpose-built support for software developers. The available tools range from those designed for a particular purpose, such as the development of test data, through to integrated applications that can be used for one or more of the phases of the development process.

CASE tools can offer the developer a number of benefits both during the development phase and after implementation. As seen, the use of CASE tools allows the developer to produce a more uniform end product. Developers who use templates to produce elements such as screens and reports find that it takes less effort to conform to the design rules that have been incorporated into the template than to create their own design. Software that has been designed with the aid of CASE tools will generally be easier to maintain as the documentation will again be more consistent. When a number of developers use the same CASE tools on a project, these tools can enhance the communication between team members, as all members can have access to all relevant documentation as they need it. These advantages will also lead to a decrease in development time. Some CASE tools themselves can save time by automating a number of tasks. A final benefit of CASE tools is their contribution to improving system quality.

Exercise 5.6

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Software development is always accompanied by the production of ___________. These may be classed as ___________ documents and ___________ documents. Documents such as _______ , ___________ and ___________ are produced during development, so they are called ___________ documentation. Manuals for the user such as ___________, ___________ and ___________ are produced for use with the ___________ as it is being used. These documents are called ___________ documentation. The first document a user is most likely to use is a(n) ___________ manual.

2 Define the term 'product documentation'. Name three documents that fall into this category. Use your examples to illustrate your answer.

3 Describe the purpose of user documentation. Explain the differences between the two types of user documentation.

4 Two of the algorithm description methods mentioned in this section are graphical (flowcharts and Nassi-Schneiderman diagrams) and two are text-based (pseudocode and structured English). What advantages does a graphical algorithm description have over a text-based system and what are the disadvantages?

5 An algorithm description is independent of both the implementation language and the computer system used for the implementation. What advantages does this have for the programmer?

6 Give reasons why a program's code listing would be useful after its implementation.

7 Explain the role that formatting plays in the understanding of a program listing.

8 Reformat the following program so that its structure becomes clear:

```
on askQuestion ask 'Please enter X, Y or Z.' if it is not
emptythen if it is 'X' then do Xmenuitem else if it is 'Y'
then do Ymenuitem else if it is 'Z' do Zmenuitem else answer
'Please enter only X,Y or Z' do askQuestion endif end
askQuestion
```

9 Explain the differences between internal and external documentation. Give an example of each type of documentation.

10 Define the term 'intrinsic documentation' when applied to a program listing. What are the advantages of using intrinsic documentation?

11 Most high-level languages allow the programmer to insert comments or remarks that are ignored by the compiler. What is the purpose of these statements?

12 Write down the features that should be present in an installation guide, giving reasons for their inclusion.

13 Explain why a well-presented installation guide is important to the user.

14 Examine the installation guide for an application of your choice. Name the application and list the features you like about the guide and those features you think could be improved, giving reasons.

15 Install a software application onto a computer system using the installation guide. What improvements, if any, would you make to the guide? Justify your answer.

16 Using the appropriate computer applications, design an installation guide for a fictitious software application. Have a non-computer user evaluate your guide for ease of reading and clarity of language. What improvements could you make to the guide?

17 Explain the main purposes for providing a user manual with a software application.

18 Examine and compare two different software user manuals. Describe their similarities and their differences.

19 Obtain a user manual for a software application that you use. Comment on its ease of use, the language used and the manner in which it is presented. What changes would you make to the manual in order to improve it?

20 A programmer has created a program for small children to practise their multiplication tables. Design a 'user manual' for this fictitious program. Use the appropriate software applications to produce your manual. Use a word processor for the task.

21 Explain the importance of a trouble-shooting guide.

22 Describe the features contained in a good trouble-shooting guide.

23 A trouble-shooting guide may illustrate the steps to solving a problem as a flowchart. Draw a flowchart that describes the steps to be taken if a file does not load successfully from a disk. Use an appropriate CASE tool or application to help in this task.

24 For a software application of your choice, examine the trouble-shooting guide. Describe the features of the guide that you like and those that you dislike. Give reasons for your answer.

25 A simple word processor can open a disk file, save a file to disk and allows the user to type text into a document. List the possible problems a user could encounter when using this program. From your list of problems, construct a trouble-shooting guide for this program. Use the appropriate computer software to produce your guide.

26 What is the purpose of keeping documents such as data dictionaries, algorithm descriptions and program source codes?

27 Examine the user documentation for a particular software application. Name each document in the set, commenting on ease of use. What improvements would you recommend?

28 Choose a commercial software application and create a tutorial that explains one function of that application. Use appropriate computer software to help you create the tutorial.

29 Compare the user manuals from a number of commercial applications from different software publishers. Comment on the layout of each of the manuals and the style of language. What are the similarities of, and differences between, the manuals?

30 Explain how CASE tools and application software can help the developer with the process of documenting a program.

Hardware environment to enable implementation of the software solution

Software forms only a part of the whole computer system. Hardware, personnel, data, procedures and other software have to be integrated with the designed solution in order for it to function effectively. During the system design process, the needs of the system are identified and the various elements required for that system to function are identified. Thus, when a custom software component is designed, the hardware and operating system requirements are known. Application software design is slightly different in this respect, as the hardware components do not form a part of the design brief. However, application software will have a number of performance criteria to meet, which provide a basis for a minimum hardware requirement and a minimum operating system requirement.

Hardware requirements

Software solutions are always developed with some form of hardware configuration in mind. Hardware needs are often dictated by the nature of the problem to be solved or by the speed with which processing must take place. For example, a program that has to process a great deal of graphical information, such as real-time video, will require a sophisticated computer with a large memory, mass storage device and a fast processor. A program that has to process text that has been input by a user will not require the same degree of power.

Since the purpose of all programs is to take some form of data as input, process it and output a result, suitable peripherals will also appear in a list of hardware requirements. For example, a word-processing program is all but useless unless its result is output to a hard copy device such as a printer. Thus a printer would form part of the hardware requirements of a word processor.

Minimum configuration

Each software title, whether custom-built or off-the-shelf, will require a minimum hardware configuration for correct operation. Some of these requirements will come from the purpose of the software, others from the system being used to convert the source code into executable code. The minimum hardware needs of an application may come from a performance requirement, for example the ability of a game to process graphical data in real time. Other hardware needs come from the requirement that the software perform a specialised task, such as controlling a manufacturing robot. Further hardware needs emanate from the processor required to execute the object code; for example, a program compiled for a Widget Processor IV may not run on a Widget Processor III, as their instruction sets are slightly different. Software also requires a certain amount of primary storage for it to function; this will add factors such as a minimum memory size for the hardware.

The minimum hardware required for a software application to successfully operate relates to the following factors:

- processor type and speed
- primary storage (RAM) available
- specific input and output devices
- secondary storage size and type
- word size of the processor (the number of bits able to be processed in one machine cycle)
- a minimum operating system (this may be a version of the operating system or it may be a particular operating system with certain utilities, extensions or drivers added).

Possible additional hardware

In addition to the minimum hardware requirements, many software titles will function more efficiently or in a greater variety of ways if optional hardware items are added to the minimum required. The additional hardware may consist of increases in the minimum requirements, for example a faster processor, or it may be an optional peripheral. For example, a scanner is not an essential part of a word-processing program. However, the capabilities of a word processor are enhanced if a scanner and optical character recognition software are added to the basic system.

Appropriate drivers or extensions

In order for a software application to function correctly, it must be able to communicate with the rest of the system. In many cases the application programmer does not write the subprograms that provide this communication channel, as they come from the operating system or additions to that operating system. The subprograms that manage communication with peripheral devices such as printers, mice, keyboards and scanners are called **drivers**. If a program is to communicate correctly with a particular device, the driver for that device must be available for use.

The operating system may also contain other subprograms that do not manage communication with a device, but provide an enhanced function to the operating system. These subprograms are known as **extensions**, as they extend the capabilities of the operating system. For example, networking capabilities may not be built into the operating system, but extensions can be produced that allow the computer to work on a network and to share files across the network.

Application or custom software may need to access extensions or drivers. In this case, the subprograms necessary for the software to function may need to be included as an optional part of the installation process. This gives the installer an opportunity to load these extensions if necessary.

Emerging technologies

Software developers are always faced with the challenge of new and evolving technologies. A software engineer has to be up-to-date with the developments in both hardware and software, as each can have an effect on the development process.

Hardware

Hardware developments can be viewed in a number of different contexts. The first concerns increases in performance, which may be achieved by an increase in the speed of the processor or the size of the word that can be processed or by adjustment of the processor instruction set. Increases in the speed and capacity of primary storage will also have an effect on the software developer, as will the availability of primary storage as a part of the processor. Developments in the area of peripheral devices can assist the developer to produce a more intuitive interface for the user or to automate part of the interface. Miniaturisation of components and a decrease in power requirements of hardware items make them more suitable for portable use.

It is impossible to predict even the short-term developments in hardware that will impact on software developers. However, developers must be willing and able to use those hardware items that are appropriate for a solution or that can add extra functionality to a pre-existing solution.

Software

Developments in the area of software also encompass a number of issues, ranging from improvements in operating systems and utilities through to new applications brought about by increases in the speed and processing power of the hardware.

Developments in operating systems and utilities are based on improving the user interface to make it more intuitive, adding extra useful functions and improving communication between people, hardware and the environment. These developments will also bring problems of compatibility of existing software with new technologies. Developers often find that a small change in an operating system, or some other software component that is essential for the running of a program, will cause problems with an existing or developing solution.

As technology improves, the programming environments that the developer has to work with also improve. Thus a programmer learning a particular language today may find that, in a short period of time, that language will cease to be useful. Thus we, as software developers, need to develop those principles and attitudes that can be applied to technologies that have not yet been developed. Twenty to thirty years ago programming languages accessible to the general computer user were fairly simple by today's standards. What languages will be like in twenty to thirty years time is anyone's guess. However, the design processes that developers employ—defining the problem and designing, implementing, testing and evaluating the solution—will still be relevant regardless of the direction that software development takes.

The effect of emerging technologies

Software will, quite naturally, be designed to take the greatest advantage of the new possibilities generated by developments in hardware and software. However, developers have a responsibility to ensure that these advantages are available to the greatest number of people possible. Thus developers have to use their social conscience when undertaking a development project to make sure that the software produced provides the greatest benefit to the greatest number of people.

New technology impacts in a number of ways. The most evident for software developers are the effects on the environment and the effects the new technology has on the software development process.

The effect on the human environment

The effect on the environment of any new or emerging technology can be viewed from a number of perspectives, ranging from an individual perspective through to a global one.

For the individual, new technology can lead to advantages such as a better standard of living, access to new forms of entertainment and information, and the ability to perform tasks that were previously impossible. On the other side of the coin, factors such as a change in employment or services can impact greatly on an individual.

The new and emerging technologies will also impact on a local, national and global scale. These impacts will vary from economic effects through to environmental effects.

The effect on the development process

As already seen, the development process itself is subject to changes brought about by the improvement of existing technologies or the introduction of new ones. In the past, the move from transistor technology to integrated circuit technology allowed software developers to use cheaper and more powerful computers in their work. This change in technology brought with it a new generation of languages that allowed people with little or no formal training in programming to create very sophisticated solutions to problems. As integrated circuit technology developed further, the computer became more intuitive and is gaining acceptance as a normal part of life.

As developers use the power of the new technologies, the manner in which software is created evolves. The same basic framework of problem solution still exists, but the technology allows some of the tasks to be performed in a very different way now from the one that was used earlier. For example, the cheapening of computing power has made it almost mandatory to provide online assistance rather than printed manuals. Even the printed manuals are now being distributed in electronic form rather than on paper.

As the technology steadily moves forward, the tools and skills developers use will be continually updated to use the new features offered by it.

Exercise 5.7

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

All software needs ________ hardware and ________ so that it can operate effectively. The ________ hardware requirements will specify the type of processor, its minimum ________ amount of ________ required and any necessary ________ devices. The software will also require a minimum standard of ________ system and will specify any ________ or ________ that are needed. This list of ________ specifications is often included in the ________ manual supplied with the software.

2 Examine the minimum hardware and software requirements for an application package. Explain, in terms of the purpose of the software, why these requirements have been chosen.

3 Describe the optional peripheral devices that could be used with the application you described in question 2. Explain how these devices can improve or extend the functions of the software.

4 Explain how the creation of Web browsers such as Mosaic, Internet Explorer and Netscape has had an effect on individual students as well as the whole population of your school.

5 Explain the effect that the wider availability of increasingly more powerful computers has had on the development of software. Give some examples to illustrate your answer.

Team Activity

Choose a simple game, such as two-up, and develop a working program to play the game. The program must be fully documented, with CASE tools and/or application software being used where appropriate. Particular attention must be paid to the design of the user interface, testing and documentation.

Review exercises

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

The design of a screen has to incorporate the needs of the __________ data and the __________ data as well as providing __________ for the user. A programmer can assist the user by providing __________ that can be employed if the user needs it. These days most applications provide __________ help as it can be easily __________, especially if a number of __________ use the software over a network. The user documentation is also known as __________ documentation as it is referred to as the program is being used. Maintainers, however, are usually more interested in the __________ documentation that is produced while the program is being __________.

2 Choose the alternative, A, B, C or D, that best answers the question:

a A metalanguage is used to
   A write a computer program
   B find errors in a program
   C describe the syntax of a language
   D describe an algorithm

b When a program uses sequential access to process data
   A an index is used to access the data item
   B each data item is accessed during a loop
   C the data items are read from disk in sequence
   D the data items are processed randomly

c All processing of data is carried out by using
   A special registers in the CPU
   B special locations in main memory
   C the control unit
   D the external data bus

d The step in the translation process in which the structure of the statement is examined is
   A scanning
   B parsing
   C type checking
   D compiling

e A running program stops execution without warning. This is an example of
   A a syntax error
   B a run time error
   C a logic error
   D a arithmetic error

3 A school is designing a swimming carnival program that uses a digital video camera to determine the places of the competitors. Describe the screen elements that should be present for this program to work properly.

4 Examine the syntax for a binary selection in two different programming languages. Describe the syntax for each using either BNF or a railroad diagram. Using your diagram, describe the similarities in, and the differences between, the way in which the selection has been defined in each language.

5 Explain the purpose of process documentation in the maintenance of a software solution. Explain why it is necessary for developers to create process documentation that is easy to read and follow at a later date.

6 Describe the steps a processor may use to exchange the values stored in memory locations 254 and 709. Illustrate your answer with a diagram.

7 Even though a coded program's algorithm has been checked, errors in logic may still occur. Describe one cause of a logic error in a coded program and give an example to illustrate your answer.

8 Design and create a reference card for the word-processing program you use at school. The card should occupy only one side of an A4 sheet and contain screen shots of the function buttons described on the card. The card should be created using an appropriate application or CASE tool.

9 Investigate the developments that have occurred in hardware during the past year. How have these developments affected the development of software?

10 Describe the ways in which the development of Internet banking software has affected the individual, the banks and the community.

Chapter summary

- An effective user interface is important to the success of a software solution.
- Screen design must take into account the types of data to be displayed, the target audience, user assistance and consistency between screens.
- Popular approaches that help with screen design include simple graphics programs, object-oriented programming languages and other development systems that include an integrated report and screen design systems.
- Help screens should be designed meet the needs of the target audience.
- Consistency in screen design allows users to anticipate actions and the placement of items.
- Metalanguages are used to describe the syntax of programming languages.
- Syntax structure diagrams show syntax graphically. BNF and EBNF are text-based metalanguages.
- Choice of a language may require a change to the way in which the algorithm is structured.
- A processor consists of an arithmetic and logical unit (ALU), registers and a control unit.
- The ALU is responsible for carrying out the processes.
- The registers are used to store instruction locations and the results of calculations.
- The control unit coordinates the processes, ensuring they are carried out in the correct order.
- Machine code instructions contain bits that represent the type of instruction to be carried out, the registers to use for the calculation and the location of the data items to be processed.
- Processor instruction sets cover arithmetic, logic, branching, movement of data, comparison and the use of subprograms. Some instruction sets will contain further specialised instructions.
- Registers are memory locations built into the processor.
- During the fetch–execute cycle, a register, known as the program counter, is used to hold the location of the next instruction to be fetched from main memory.
- The stack pointer is used to keep track of the instruction to return to after a subprogram has been executed.
- Some of the subprograms used by a program will come from outside the written code.
- The source code, the human understandable form of a program, has to be converted into object code, the machine understandable form of the program, by a translation system.
- A compiler converts all the source code to object code which is then stored for later execution.
- An interpreter translates each line and executes it before the next line is translated.
- An incremental compiler translates the commonly executed routines and stores them. The program is then interpreted, but when these common routines are needed they are executed from the stored code rather than being re-translated.
- The steps of the translation process common to all methods are lexical analysis (scanning), syntactic analysis (parsing), semantic analysis (type checking) and code generation. A compiler may use an optimiser to help make the program run more efficiently.
- The scanner reads the source code one character at a time, using the rules of the language to create recognisable language elements that are then represented as a code called a token.
- The tokens are then passed on to the syntactic analyser (parser) that arranges the tokens in a way that allows the compiler to understand the logic of the program.
- If tokens cannot be placed on a parse tree, the translator flags an error.
- After parsing, the tokens are sent to the type checker which detects the data types within the tokens and incompatible operations between different data types.
- The code generator converts the tokens into the object code by working through the parse trees.
- A linker is used by a compiler to join the translated code to code from other sources such as the operating system.

Chapter summary

- A compiler adds a special program known as a loader to the object code. The loader's task is to enable the program to run in a different location each time it is executed.
- Compiled code is often optimised so that it works more efficiently.
- Compiled code runs faster than code translated by the other methods, is harder to modify, occupies less memory and hides the original algorithms. The main disadvantage of compiled code is that run-time errors are not apparent until the code is executed.
- Interpreters allow all errors to become apparent as the program is being run. However, the code is visible to the user and may be easily modified or illegally used.
- Incremental compilation retains for developers the advantage of an interpreter that all errors become apparent without complete translation of the code. The program runs faster than for an interpreter, but slower than for a compiler.
- A number of different techniques will help in the development process. These include restricting each subroutine to one logical task and the use of stubs during development.
- Flags, the isolation of errors and debugging output statements can also assist during the testing stage.
- Programs should be written to aid future maintenance.
- Errors in programs fall into one of three categories: syntax errors, logic errors and run-time errors.
- Syntax errors occur where a program structure in a language does not follow the rules of the language.
- Logic errors are errors in processing where the program flow does not follow the correct sequence of instructions to solve the problem.
- Run-time errors are errors that occur during the execution of a program.
- Errors in the program are detected by using a number of tools..
- Once an error has been detected, it is corrected and the program is again tested for correct working.
- Some of the more common debugging tools and techniques are the use of breakpoints, program traces and single line stepping. Resetting variables can also help with debugging.
- Software documentation tells us what the software is to do, how it performs its tasks and what a user needs to do to run a program.
- Documentation falls into two categories: process documentation and product documentation.
- Process documentation is all documentation that relates to the development process.
- The coded solution should be self-documenting to make the maintenance process easier.
- Product documentation relates to the use of the software.
- Product documentation includes installation manuals, user manuals, trouble-shooting guides, reference cards, help manuals and tutorials.
- Some product documentation can be provided online.
- Documentation required for maintenance includes the algorithms and source code, data dictionaries, and screen and report designs.
- CASE tools or application programs can accomplish many of the documentation tasks associated with software development.
- A software solution will require a minimum hardware and software combination to function correctly.
- Emerging and new technologies can provide the software developer with expanded opportunities in both development and the final product.

chapter 6 Testing and evaluation of software solutions

Outcomes

A student:

- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- constructs software solutions that address legal, social and ethical issues (H 3.2)
- applies appropriate development methods to solve software problems (H 4.2)
- applies a modular approach to implement well-structured software solutions and evaluates their effectiveness (H 4.3)
- applies project management techniques to maximise the productivity of the software development (H 5.1)
- creates and justifies the need for the various types of documentation required for a software solution (H 5.2)
- selects and applies appropriate software to facilitate the design and development of software solutions (H 5.3)
- assesses the relationship between the roles of people involved in the software development cycle (H 6.1)
- communicates the processes involved in a software solution to an inexperienced user (H 6.2)
- uses a collaborative approach during the software development cycle (H 6.3)
- develops effective user interfaces, in consultation with appropriate people (H 6.4)

Students learn about:

Testing the software solution

- comparison of the solution with the original design specifications
- generating relevant test data for complex solutions
- levels of testing
  - unit or module
  - program
  - system
- The use of live test data to test the complete solution:
  - larger file sizes
  - mix of transaction types
  - response times
  - volume data
  - interfaces between modules
  - comparison with program test data
- benchmarking
- quality assurance

Reporting on the testing process

- documentation of the test data and output produced
  - use of CASE tools
- communication with those for whom the solution has been developed, including:
  - test results
  - comparison with the original design specifications

Students learn to:

- differentiate between systems and program test data
- test their solution with the test data created at the design stage, comparing actual output with that expected
- demonstrate the features of a new system to users, facilitating open discussion and evaluation

Testing the software solution

In the Preliminary Course it was noted that there are two aspects to testing a software solution: validation and verification. Validation is the process of comparing the solution with the design specifications; verification is the process of ensuring that the software performs its functions correctly. If the software solution is to be successful, it must pass both types of test.

I am writing a test sentence.
Siam fighting atomic sentry.
SCRATCH! SCRATCH!
I am writing a test sentence.
Ian is riding a taste sensation.
SCRATCH! SCRATCH!
I am writing a test sentence!!
I am writing a test sentence!
SCRATCH! SCRATCH!
Catching on?
Egg freckles?
SCRATCH! SCRATCH!

Figure 6.1 Software testing can be a long and frustrating experience.

Comparison of the solution with the original design specifications

Design specifications should be written for a software solution in a form that provides a set of performance criteria. In this respect a set of software specifications is no different from the set of features you would have in mind when contemplating the purchase of an expensive item such as a house or car. When you go looking for the item, you evaluate each of those on offer, choosing the one that fits all or most of your criteria. Part of the process of choosing a car is to 'test drive' it to ensure that it performs in the way you expect it to. The same is true for software. There is no gain for either the client or the developer in producing software that does not perform the tasks the client requires.

When the software specifications are written in a form that provides criteria for the measurement of performance, they help both the software team and client evaluate the performance of the end product. It pays to review specifications before software construction takes place so that they are as clear as they can possibly be. The following guidelines can be used to review the detailed specifications:

- Ensure that the specifications are written in terms of measurable outcomes.
- Clarify any vague terms (e.g. terms such as 'most', 'some', 'sometimes' and 'usually').
- Avoid the use of vague verbs such as 'processed', 'eliminated' and 'handled', as they may be interpreted in more than one way.
- Look for and clarify any ambiguous statements (e.g. 'The data input module sends the parameter to the backup and recovery module and its flag is set').

Figure 6.2 We test drive a car to make sure that it performs as expected. We must do the same with software.

- Incomplete lists of items should be avoided. If this is not possible, then all list items should be clearly understood. Terms such as 'etc', 'and so on' and 'such as' are used as indicators of incomplete lists and should be avoided.
- Question the use of persuasive terms such as 'certainly', 'clearly' and 'obviously'. Determine why these terms are used, and if there is no real reason for them, the specification should be rewritten.
- Calculations should be accompanied by examples.
- Pictures and diagrams should be used to clarify structure, rather than relying on descriptions.

If guidelines such as these are followed in the specification document, they ensure that evaluation of the product can be made in an objective way. For example, each of the requirements could be represented on a grid and ticked off as to whether the requirement was fulfilled in full, in part or not fulfilled. The grid can then be used as a basis for any modifications that may be needed.

The process of validation involves the use of the software in a 'real' situation with live data (real data items) so that its performance can be compared with the set of specifications under conditions that are as close as possible to those that will be encountered once it has been fully implemented. This process needs to be properly documented, as recommendations made at this stage have to be incorporated into the final product.

Figure 6.3 Validation takes place under conditions that are as close as possible to those of the final implementation.

Generating relevant test data for complex solutions

In discussing testing, the emphasis so far has been on checking that the program or algorithm performs to the required specifications. There has been no discussion of how to create test data that can locate all possible operational faults. Unfortunately there are often so many variables that complete testing is an expensive and/or time-consuming impossibility.

Many operational factors may be outside the control of the programmer, or unable to be foreseen. These factors include inappropriate inputs, variations in hardware, changes or differences in operating systems and changing technologies. It is not possible to foresee inappropriate inputs by the user; for example, the use of a particular key combination may be intercepted by the operating system or data of the wrong type may be entered. Variations in hardware items because of different manufacturers, modified chips or other reasons can mean that machines, which on first inspection appear identical, may function slightly differently. As operating systems are updated or changed, the problem of compatibility with existing software also becomes evident.

Changing technology also poses a problem, as newer systems containing processors or operating systems, which are backward compatible in most respects, may contain minor differences that cause errors during program execution.

All of these factors can limit the reliability of testing. The programmer can test the program only within the bounds that are set during the analysis stage of development.

A further hurdle in program development occurs with the increasing size of applications. As more and more features are required from a program, its complexity increases greatly. It may be almost impossible to create test data to check every path through the program, because of the multiplying effect of each decision. Modularisation of programs means that data can be created to test each of the modules; however, it is also likely that an unforeseen output from one module will cause problems in other modules in the program.

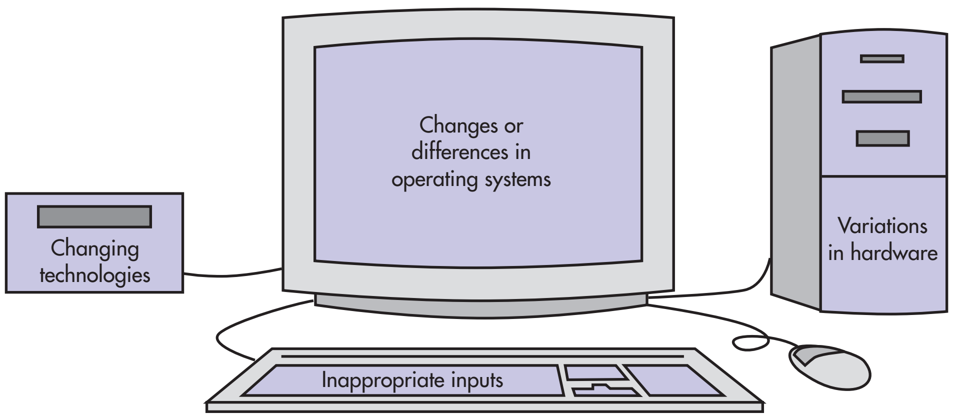
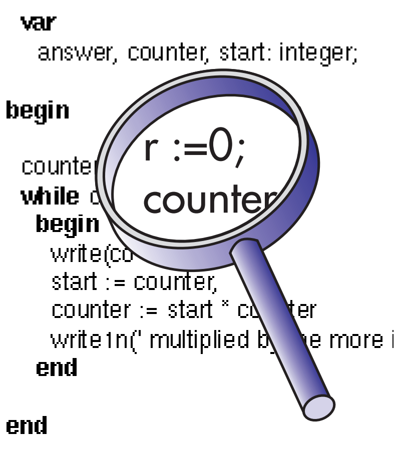


Figure 6.4 Many operational factors are outside the control of the programmer.

There are various techniques that can be used to create test data that will detect most of the operational errors while ensuring that the time and effort expended on testing is not out of proportion to the other aspects of development.

When a small module is designed, it may be possible to generate test data cases that cover all the possible paths through the algorithm. However, as modules become more complex or are combined to form a more complex module, testing of every path is not a viable option. In these cases, the program development team choose test data items to uncover anticipated problems.

Programmers use a number of different techniques to test the internal workings of a module or program. These testing procedures are known as **white box testing**, as they concentrate on the internal working of the module. The most common white box testing procedures in use are statement coverage testing, decision–condition coverage testing, multiple decision–condition coverage testing and exhaustive condition decision–condition coverage testing.



```
var
  answer, counter, start: integer;

begin
  counte
  while c
   begin
    write(co
    start := counter,
    counter := start * c      ter
    write1n(' multiplied b    e more i
   end

end
```

Figure 6.5 White box testing examines the workings of a module.

The following algorithm structure will be used to illustrate the application of each method for creating test data elements.

```
BEGIN sample_algorithm
    WHILE (condition 1)
        code section A
        IF (condition 2)
        THEN
            WHILE (condition 3) AND (condition 4)
                code section B
            ENDWHILE
        ELSE
            IF (condition 5)
            THEN
                code section C
            ELSE
                code section D
            ENDIF
        ENDIF
    ENDWHILE
END sample_algorithm
```

Statement coverage testing involves the execution of every statement in the module. When choosing test data items for statement coverage testing, developers choose the data items so that each statement in the module will have been executed at least once by the time all data elements have been processed.

To perform a statement coverage test for the sample algorithm, three sets of test data elements would be created. The sets would contain values that make:

- Set 1—all conditions true to pass control through code sections A and B.
- Set 2—condition 1 true, condition 2 false and conditions 3, 4 and 5 true to pass control through code sections A and C.
- Set 3—condition 1 true, condition 2 false, conditions 3 and 4 true, with condition 5 false in order to pass control through code sections A and D.

When all test data elements have been used, each of the code sections will have been executed at least once. (Notice that code section A will have been executed three times.)

Decision–condition coverage testing involves full statement coverage as in the previous method, but also tests the execution of each decision in control structures with its 'true' and 'false' at least once. Thus the test data elements can be grouped in sets that contain elements that fulfil the following conditions:

- Set 1—all conditions true to pass control through code sections A and B.
- Set 2—condition 1 true, condition 2 false and conditions 3, 4 and 5 true to pass control through code sections A and C.
- Set 3—condition 1 true, condition 2 false, conditions 3 and 4 true, with condition 5 false in order to pass control through code sections A and D.
- Set 4—all conditions true except condition 1 which is false will pass control to the end without executing any sections of code. This tests termination of the first WHILE loop. (This set is only one of a number that tests this part, as the values of conditions 2 to 5 are irrelevant to the path taken.)
- Set 5—conditions 1, 2 and 5 are true, with 3 and 4 false to test the second WHILE loop.

Adding the two extra sets of data increases the testing to a check of all the decisions, but this does not test all possible paths. For example, what happens when condition 3 is true and condition 4 is false has not been checked.

Multiple decision–condition coverage testing extends the process of decision–condition coverage testing to all possible combinations within multiple decisions. In the example, we have not yet tested what happens when condition 3 is true, condition 4 is false and the other three conditions (conditions 1, 2 and 5) are true. Neither have we tested that decision when condition 3 is false and condition 4 is true (again while conditions 1, 2 and 5 are true). Thus the following seven sets of data are created to test the algorithm:

- Set 1—all conditions true to pass control through code sections A and B.
- Set 2—condition 1 true, condition 2 false and conditions 3, 4 and 5 true to pass control through code sections A and C.
- Set 3—condition 1 true, condition 2 false, conditions 3 and 4 true, with condition 5 false in order to pass control through code sections A and D.
- Set 4—all conditions true except condition 1 which is false will pass control to the end without executing any sections of code. This tests termination of the first WHILE loop. (This set is only one of a number that tests this part, as the values of conditions 2 to 5 are irrelevant to the path taken.)
- Set 5—conditions 1, 2 and 5 are true, with 3 and 4 false to test the second WHILE loop.
- Set 6—conditions 1, 2 and 5 are true, with condition 3 being true and condition 4 being false to test the second WHILE loop with one of the two combinations in the test that have not yet been tried.

- Set 7—conditions 1, 2 and 5 are true, with condition 3 being false and condition 4 being true to test the second WHILE loop with the other of the two combinations in the test that have not yet been tried.

Exhaustive condition decision–condition coverage testing involves creating sets of test data that examine what happens with all possible combinations of true and false for each condition. In the example, since there are five different conditions, each with two possibilities, there are 32 (2^5) different combinations. These are listed in Table 6.1.

The greatest advantage in creating and using a set of test data that exhaustively tests an algorithm is that it will bring out any unexpected errors that may occur with unusual data combinations. It must be remembered at all times that the aim of testing is to uncover errors rather than to show that the program works.

Exhaustive testing is the most thorough of the testing processes; however, it is most effective with algorithms and programs that have a small number of decisions. This again highlights the need to create modular programs in which each of the modules contains a relatively small number of decisions; that is, they perform one logical step of the whole process. This is listed in Table 6.1 on the the next page.

The second form of testing concentrates on the inputs of a module and the resulting outputs, without tracing the execution of any of the statements within the module. This form of testing is known as **black box testing**. It is given this name as the module is treated like a 'black box' that has a 'magical' property of transforming one thing (the input) into another (the output). The internal workings of the 'black box' are irrelevant as long as the 'box' does the job. Test data elements that could be responsible for a failure have to be chosen for this type of testing.

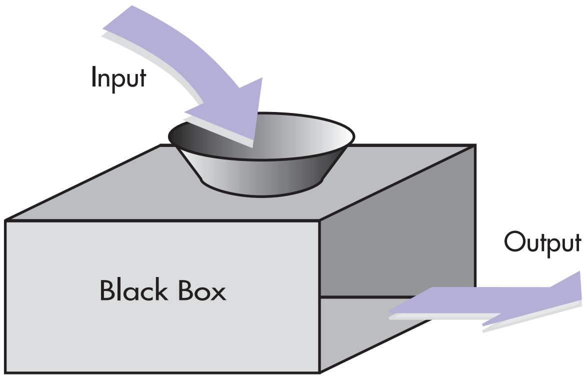


Figure 6.6 Black box testing treats the module as a 'magic box' that performs some kind of process to turn inputs into outputs.

There are two forms of black box testing: boundary analysis and equivalence partitioning.

Boundary analysis involves choosing test data elements that are representative of either side of a boundary where the effects of processing are different.

For example, a program used to calculate a discount from a store has to work as follows. 'There is no discount for an order of less than 10 items, a 10% discount off the total bill is given for 11 to 100 items and a 15% discount off the bill is given for orders above 100 items.' The boundaries in this program are for 10 and 100 items. Thus the test data set for this program would include values that are less than 10, between 10 and 100 and over 100. The output from each of these inputs would be compared with the expected outputs that have been calculated manually.

Equivalence partitioning involves breaking up the input data into groups that have the same properties. For example, we might have a database field that is to store a family name. Only combinations of a limited set of characters are allowed as family names. (The character set for this data type would consist of upper- and lower-case letters, the hyphen and the apostrophe.) Thus we would have one partition of input data that can be processed as a family name, that is, any combination of characters from the limited set. Input data items for this field that contained any characters not in our allowed set would form a second partition that would be processed as containing illegal characters.

When sets of data are created to test a module, representative combinations of characters are chosen from each of the partitions. For our example, some combinations that contained only the allowable characters described in the previous paragraph and others that contained one or more illegal characters would be chosen. It would be expected that the data items

Data set	Condition 1	Condition 2	Condition 3	Condition 4	Condition 5
1	True	True	True	True	True
2	True	True	True	True	False
3	True	True	True	False	True
4	True	True	True	False	False
5	True	True	False	True	True
6	True	True	False	True	False
7	True	True	False	False	True
8	True	True	False	False	False
9	True	False	True	True	True
10	True	False	True	True	False
11	True	False	True	False	True
12	True	False	True	False	False
13	True	False	False	True	True
14	True	False	False	True	False
15	True	False	False	False	True
16	True	False	False	False	False
17	False	True	True	True	True
18	False	True	True	True	False
19	False	True	True	False	True
20	False	True	True	False	False
21	False	True	False	True	True
22	False	True	False	True	False
23	False	True	False	False	True
24	False	True	False	False	False
25	False	False	True	True	True
26	False	False	True	True	False
27	False	False	True	False	True
28	False	False	True	False	False
29	False	False	False	True	True
30	False	False	False	True	False
31	False	False	False	False	True
32	False	False	False	False	False

Table 6.1 The five conditions within the module mean that there are 2^5 combinations of true and false for them.

that consist of all legal characters would be processed as legal family names and those that contained the other characters would be rejected. Any deviation by the module from this anticipated course of action would signal that there was an error in the processing.

It should be noted that 'legal' family names need not necessarily be ones that make sense, or are even pronounceable. For example, the 'names' *ZSAWER-TRYWQ*, *sewmna'Warsk* or even -'--'- are legitimate data items for this set of test data. Examples of data items that could be used to represent the partition of illegal family names would be strings such as *Der3ginty*, *jon*s* and *^&$%#@()*. It is also interesting to note that the most likely of these data items to pick up an error is the one that contains the least predictable set of input characters.

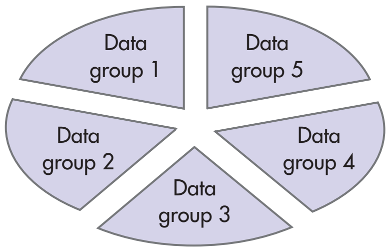


Figure 6.7 Equivalence partitioning divides data into groups that have the same properties.

Levels of testing

Two different testing procedures can be employed to test a modular program; these are black box testing and white box testing.

The concept of black box testing treats each of the modules as a black box that takes certain inputs, creating the wanted outputs. The manner in which the black box achieves its results is irrelevant, as long as the desired outputs are obtained. Black box testing is applied in order to test whether inputs are acceptable and whether the desired outputs are achieved.

White box testing exposes the details of the module and is concerned with the correct functioning of that module. Test data for this type of testing needs to check all paths through the system. White box testing is usually applied to each of the individual modules.

By combining the two testing methods, complex programs can be well tested, the white box testing being used to check each of the modules and the black box testing being used to test their interaction.

Unit or module testing

Unit testing (or **module testing**) treats each of the modules as a stand-alone application that does not require any of the other components of the program to function. The module is tested with an appropriate test data set in order to uncover processing faults. The test data elements have been constructed prior to the testing procedure to highlight any problems with processing.

There are a number of different types of error that might occur within a module or unit. These can be broadly classified in the following manner.

- Arithmetic errors are those in which calculations are not performed correctly, or illegal operations such as divisions by zero occur.
- Comparison errors are those in which comparisons are attempted on different data types or the choice of multiple comparisons is incorrect. For example, an 'AND' has been used in a comparison where 'OR' should have been used.
- Control logic errors are those in which looping or branching is not performed correctly. These errors will most likely be due to a wrong choice of comparison, boundary values or, in the case of loops, incorrect initialisation or termination.
- Data structure errors are those that occur in the use of data structures that are internal to the module. These would include the use of flags, counters and accumulators (variables used within the module to total amounts).
- Input and output errors are those that may occur when reading and writing files. Testing of this aspect of a module ensures that the data items received from a file and/or output to a file are those that are expected.
- Interface error detection focuses on the parameters that are passed from one module to the next. The parameters tested would include data items that are passed between the modules and those that are used for control, such as flags.

Modules are generally tested as white boxes, with the emphasis being on the logic, and as black boxes, with the emphasis being on the function of the module. The one set of test data may be sufficient for both forms of testing; however, it may be necessary to use two different sets of test data for the two types of test.

Program

Exhaustive tests performed on all modules of a program will not guarantee that the program will work as desired. Interaction between modules is an important aspect of program design and is often a cause of problems within the program. The process of combining the modules is known as **integration** and the testing that is carried out as part of this process is known as **integration testing**.

Once modules have been passed as being able to function correctly, they can then be assembled into a working program. It is tempting to put all the modules together to create a fully working solution as soon as possible after they have been tested. The danger with this approach is that the causes of errors brought about by the interaction between modules are hard to detect and rectify. Two approaches are taken in the assembly of a program from its component module: the top-down approach and the bottom-up approach.

In the **top-down approach**, the program driver module is first tested with stubs representing each of the lower modules. Once the driver works properly, the modules are gradually added, the program is tested at that stage, any corrections are made and tested, and then a further module is added. This process of adding modules and testing continues until the program is complete. The advantage of this approach is that any errors that are detected will be due to the most recently added module. This same approach can be taken to build large modules from a number of smaller sub-modules.

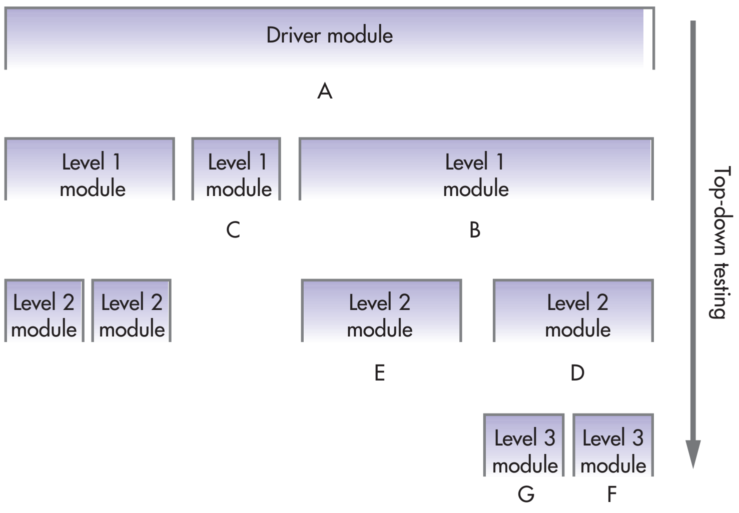


Figure 6.8 Top-down testing starts at the driver module and works down the module levels.

The **bottom-up approach** involves testing the lower-level modules first, then adding them together, one by one, and testing, modifying and re-testing the larger module until it functions correctly. Thus the program is built up from the smaller modules to a fully working program, with the driver module being the last to be added.

Once the modules have been assembled into the final product, the process of testing is still not finished. The program now has to be tested against the specifications of the user. This stage of testing is known as **function testing**. Users become actively involved at this time. During this process test cases developed in the analysis and design stages are used to measure the degree to which the program measures up to the requirements of the user. The emphasis at this stage of testing is on input and output formats, file organisation and access and the human/machine interfaces.

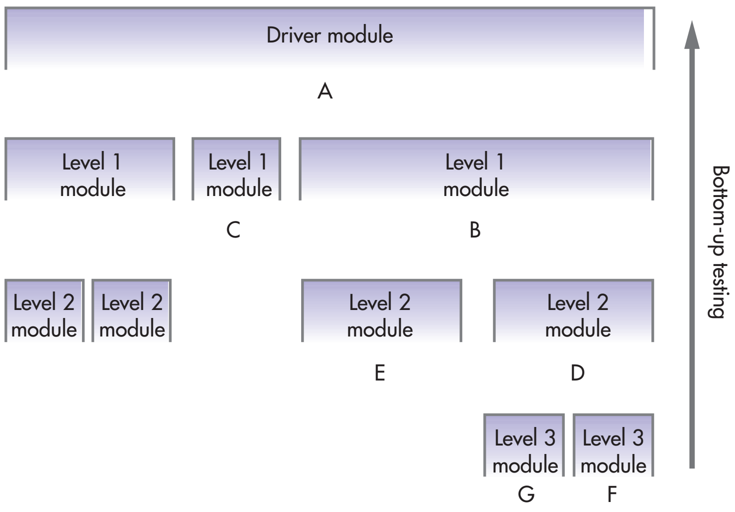


Figure 6.9 Bottom-up testing starts at the lowest-level modules, working up the levels to the driver module.

Testing does not finish once the program has been assembled. It is now time for the program to undergo **acceptance testing** (if it is a custom-built solution) or **alpha** and **beta testing** (if the software is a commercial off-the-shelf application).

Acceptance testing is a testing procedure carried out by the software users in order to determine whether the software is capable of being used in the fully operational system. Real data are used in this phase of the development process. The main purpose of acceptance testing is to ensure that the program fits into the processes and procedures of the system for which it was designed. Other aims are to gauge how the human/computer interface works with users and to detect any errors that may have been missed during the previous testing stages. Acceptance testing can also serve to train some of the users in the operation of the system.

When an off-the-shelf software title has been assembled, it would be an impossible task for all users to test the program. In this case, a two-stage equivalent of acceptance testing is used. Alpha testing is carried out in a controlled environment, with the developers watching a sample of users operating the program. Developers document any errors or usage problems as they occur. These errors and problems form the basis for further modifications to the program.

At the conclusion of alpha testing the program is passed on to a number of beta testers. These testers are given the task of trying to make the program fail under simulated real situations similar to those of the alpha testers; however, the beta testers document the problems that they find. The main difference between alpha and beta testing is the absence of developers during the process of beta testing. This allows the beta tester to use some imagination in creating situations that may cause the program to fail. Any failures in the working of the program are investigated and corrected.

System

Once the software has been tested, it can be installed onto the hardware. A further round of tests is performed to uncover any problems.

System testing can be thought of as the combination of a number of different test sequences that have been designed to fully extend the system. These tests focus on performance, recovery, security and stress.

Figure 6.10 Installation of the software on the required hardware must take place before system testing can proceed.

Performance testing is used to test the quality of the performance of the software. One of the tests performed is to gradually increase the transaction load of the system until it fails. This test gauges the transaction volume that can be handled. A second set of tests is concerned with the system's ability to recover from a fault. **Recovery testing**, as it is called, consists of a number of tests that force the system to fail and then measures the ability of the system to recover itself or the average time taken for humans to recover it. **Security testing** involves attempting to breach the security of the system. These tests will not just involve the computer aspect of security but also security within the manual procedures. **Stress testing** attempts to create situations in which the system may fail. In this process, stressful situations such as rapid reading and writing to disk or maximum main memory usage are created to determine whether the system can cope or whether it will fail. Some of these tests may overlap the performance tests.

The use of live test data to test the complete solution

During the development process, modules are tested with data that has been created by the development team. These data elements have been designed to test the operation of program modules and may have little in common with the actual data that has to be processed. It is rather like testing a car on a test track. The tests will determine how the car performs on the test track and whether modules such as the engine and brakes work. However, they will not determine how the car will perform its task in city traffic or on hundreds of kilometres of country roads. These results can be gained only by driving on ordinary roads. In the same way, a computer program or system will not reveal how it handles situations that occur in normal operation until it is subjected to the same conditions it will encounter during normal operation. The most evident of these conditions is in relation to data.

Larger file sizes

The use of real data, commonly called **live data**, will allow a number of factors to be tested under simulated operational conditions. One of the hardest conditions to simulate with manufactured data is the processing of large amounts of data or large files. The time taken to manufacture such a set of artificial test data is often out of proportion to the time that the data items will be used. Since this aspect of the program's performance has to be tested, the most sensible choice for such data comes from pre-existing system data. The outputs of such data are known already as they have been processed with the old system.

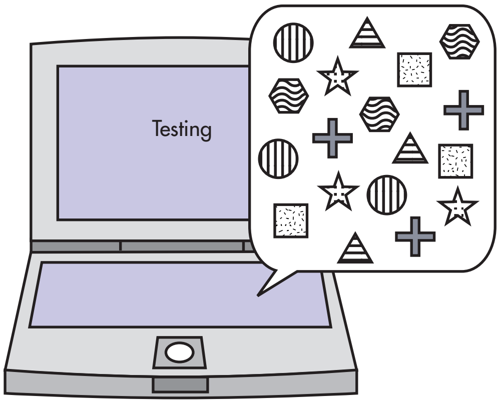


Figure 6.11 Use of live data for testing will produce a mix of typical transactions.

Mix of transaction types

The software design team, although familiar with the workings of the system, often cannot fabricate data sets that provide the mix of transactions that the operational system will encounter. Live data can provide such a mix, as the data items are a sample of real transactions.

Response times

The **response time** of an application is the time that it takes to provide a reply to data input. It is difficult to estimate the response time while the program is undergoing testing with manufactured test data items, as the purpose of these items is to uncover faults in the program. Live data items are representative of real processing situations and will, therefore, be subject to the same processing as is expected during program use. The software development team can accurately gauge the expected response times from these data items.

Volume data

The sheer volume of data that has to be processed by a program can also lead to problems. It is pointless to expect the software development team to create large sets of test data when a ready source exists. It is much more economical in both time and money to use pre-existing data from the current system for these tests. In this way, how the system will respond to a large amount of data can be measured.

Interfaces between modules

Live data provides a good test of the interfaces between the various modules of the program. The reason for this is that the live data elements are genuine representations of the kinds of data elements that will be processed in the future. If any of these data elements cause interface problems, it is reasonable to assume that these problems will also exist during use of the program.

Comparison with program test data

A comparison between test data elements and live data can highlight any shortcomings that the test data or the live data set may have had. The outputs of live data can be compared with those of the developer-created test data items to gauge the effectiveness of the test data cases.

Benchmarking

Benchmarking involves creating a set of tests that can be used to measure the speed with which a computer system will perform a particular task. The way in which the tests are structured will allow a comparison between different combinations of hardware and software. Thus the set of tests may be performed with both the old and the new systems and a comparison made between their performance. Benchmarking is a purely objective process and so subjective measurements of aspects such as user friendliness and ergonomic factors are not included in the process. The results obtained from benchmarking are often unexpected and open to different interpretations. Thus interpreting the results is just as important as obtaining them in the first place.

Figure 6.12 A benchmark serves as a reference point when surveyors want to find the elevations of points in an area.

It is important that the measurements taken are accurate and reflect a true measure of the speed of the operation and so the process is very detailed and time-consuming.

Quality assurance

One of the goals of a software development team is to produce a product that is of the highest possible quality. Quality cannot be 'added' to an existing product; it must be built into it. The process of ensuring that the end product meets the standards set for a 'quality product' is known as **quality assurance**.

It is difficult to state exactly what the properties of 'quality software' are. However, a number of attributes contribute to software quality, including:

- *clarity*—the meeting of a set of standards for the user interface, for example precise and unambiguous instructions.
- *correctness*—the consistent production of the correct output for a given set of inputs.
- *documentation*—consistent documentation and of a high standard.
- *economy*—software that is economical both of processing needs (such as main storage needs) and external resources (such as user time).
- *efficiency*—efficient production of the output.
- *flexibility*—the ability to cope with all the situations found during processing.
- *generality*—the need for software to present itself as performing a set of generalised tasks. (This means, for example, that the user interface may be set up to simulate a non-computer situation, for example a word-processor interface that is set up to resemble typing on a sheet of paper.)
- *integrity*—the ability of a system to withstand any attacks on its security, whether or not these attacks are intentional.
- *interoperability*—the ability of the software product to communicate with pre-existing software. (For example, the new software being installed on the system must be able to work with the pre-existing operating system.)
- *maintainability*—the ease with which an error in the software can be corrected.

- *modifiability*—the ease with which the software can be changed to meet new needs or circumstances.
- *modularity*—being able to replace one part without creating a completely new product. (Thus, the replacement of a peripheral, such as a manufacturing robot, will not make it necessary to completely rewrite the software. Perhaps only the robot-controlling module may need replacement.)
- *portability*—the ability of software to be executed with different hardware and software combinations. (For example, a simple Pascal program can be executed on any hardware and software combination that has the ability to translate the Pascal program and execute it.)
- *reliability*—a measure of the failure rate of software. (Thus software that rarely fails is reliable and software that continually fails is unreliable. Failure here is taken in its broadest sense to mean that the software has not performed its required task.)
- *resilience*—a measure of the software's ability to recover from an abnormal situation. (For example, how does the software recover from an unexpected finish to reading a file?)
- *reusability*—a quality built into the software at design time when an effort is made to ensure that the components being assembled for the current task can be used in future development activities.
- *testability*—how easily the software can be tested.
- *understandability*—how well the design of the software is understood. (The understandability of a program can be gauged from the quality of the technical documentation that accompanies it. For example, the interrelationships between the various components needs to be well understood.)
- *useability*—a measure of a number of aspects of importance to the user. (These are the skill and time needed to learn the program, the time required to become efficient in its use, the increase in productivity achieved through the use of the software and the attitude of the user towards the system.)
- *validity*—a measure of how the software product meets the specifications of the user.

Exercise 6.1

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.
Two aspects of software testing are ____________ and ____________. ____________ is the process of comparing the solution to the ____________ specification. ____________ is the process that ensures that the software performs its ____________ correctly. ____________ data items need to be created to test whether modules function correctly. These items are listed in a ____________ data dictionary that also lists the ____________ outputs and reasons for ____________ each data item. ____________ data items come from the existing system and are used to test whether the ____________ can handle real processing situations.

2 Describe the processes of verification and validation. What is the purpose of each of these testing procedures?

3 Write a set of design specifications for a birthday book program. This program is to be able to store people's birthdays and their contact details.

4 A module for a program has to be able to determine and output whether two input numbers are equal, or output the larger one. Create a set of test data that could be used to check the workings of this module. Don't forget to show the reason for choosing the values and the expected output for each of your test data sets.

5 Describe the processes of black box testing and white box testing. Which of these testing methods would be used with the test data you created in question 4 to test the module?

6 Statement coverage testing is to be used to test the module of a program represented by Figure 6.13. Create test data sets that could be used for this test. Describe the paths that would be taken when each of your sets is processed by the module. For example, a wage of $15 000 and a tax concession would use the path through A and E.

7 Create a set of test data for the module described in Figure 6.13 if the module is to be tested using the decision coverage method. Again describe the paths that would be taken when each of the test data sets is processed.

8 Use the algorithm in Figure 6.14 to create sets of test data that can be used to perform multiple decision–condition coverage testing on the module.

9 Use the algorithm in Figure 6.14 to create sets of test data that can be used to perform exhaustive decision–condition coverage testing on the module.

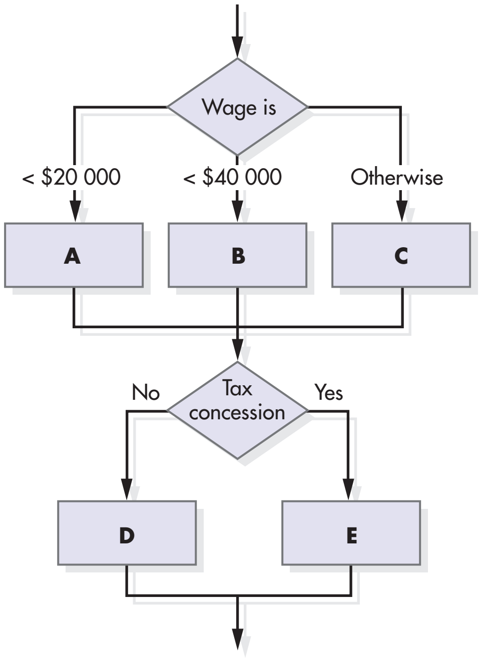


Figure 6.13

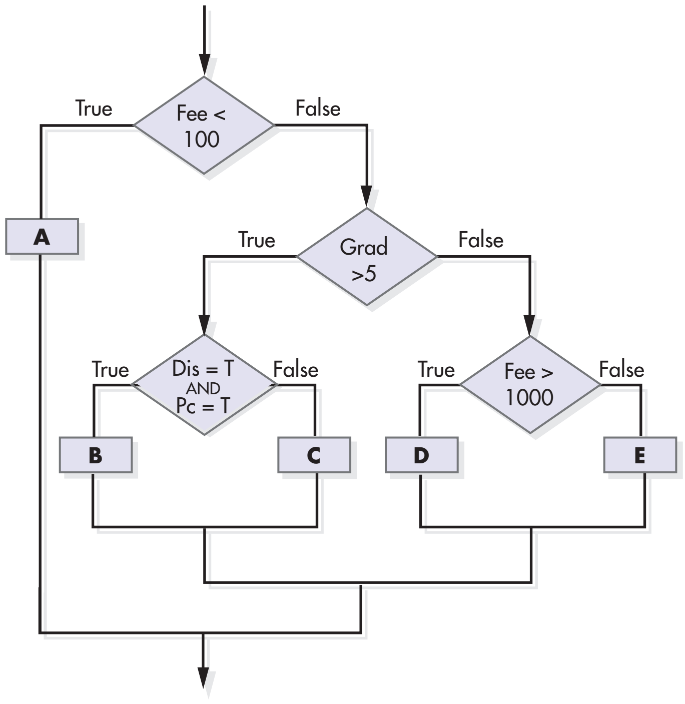


Figure 6.14

10 Create a set of boundary analysis test data items that can be used to test a module for the following description of an egg-sorting process. Your set of test data should contain the test data item, the expected output and the reason for choosing the data item. You need not worry about creating test data items to test the data validation processes within the module.

Eggs are to be graded as 'SMALL' if their weight is less than 50 grams, 'MEDIUM' if their weight is from 50 grams to less than 60 grams, 'LARGE' if their weight is from 60 grams to less than 70 grams and 'EXTRA LARGE' if their weight is 70 grams or more.

11 A module of a program processes a telephone number. Telephone numbers are restricted to strings of eight digits (e.g. 14567890) or an area code of two digits enclosed in brackets together with an eight-digit number (e.g. (01) 23456987). Create three sets of ten telephone numbers. The three sets are to be used in testing the module's acceptance or rejection of a telephone number. One of the three sets should contain eight-digit numbers, the second set should contain telephone numbers with area codes, and the third set should contain 'numbers' with some illegal characters. Name the type of test data sets that you have created.

12 A number of modules need to be tested and integrated into an application. Name and describe the two possible ways of performing this integration.

13 Describe the purpose of acceptance testing as part of the software development process. In what ways is acceptance testing different from module and integration testing? Use an example to illustrate your answer.

14 What is system testing? Why is it necessary to perform system tests after all the other tests have been completed? Describe the different aspects of the system that are tested at this time.

15 Choose a public domain program and perform a beta test on this application. Describe in detail the tests that you performed, the results of your tests and any recommendations you have for improvement. Present your answer as a word-processed report.

Reporting on the testing process

Testing needs to be a carefully controlled and executed process if it is to provide the maximum amount of information about the software. The results of the process have to be analysed with care so that nothing is overlooked. The only way that this can be achieved is if the test process is properly documented.

Documentation of the test data and output produced

Documentation of the test process involves three basic forms. The first is a **software test description** that describes the preparations for the tests, the test cases and the procedures followed. These details allow the client to determine the adequacy of the testing process. The second report is a **software test plan** that describes the test environment and the tests to be performed and provides a time frame for conducting the tests. This allows the client to review the test process. The final document is the **software test report** that provides information about the conduct of the tests and the results.

The first section of the software test description is a description of the scope of the tests. It will also briefly state the purpose of the system and the nature of the system. The procedures necessary to prepare the hardware systems, software systems and personnel for the tests follow. A description of each of the tests makes up the third part of the report. In this part are a list of the prerequisite

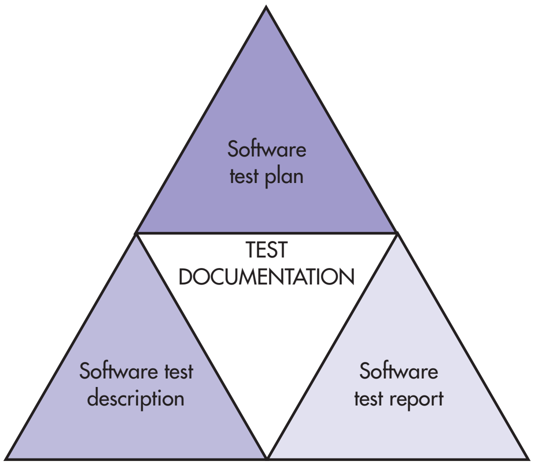
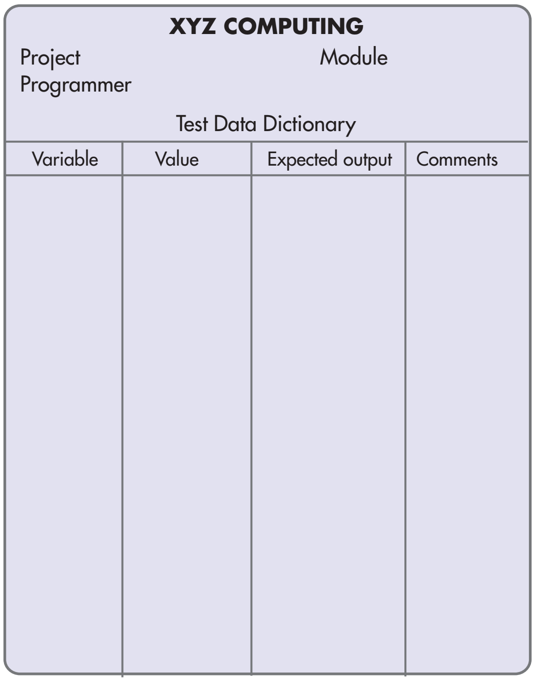


Figure 6.15 Test documentation comprises three reports: the software test plan, the software test description and the software test report.


XYZ COMPUTING

Project Module

Programmer

Test Data Dictionary

Variable	Value	Expected output	Comments

Figure 6.16 A test data dictionary will list the test data items, the expected outputs and the reasons for choosing the specific data item.

conditions, the test inputs (often in the form of a test data dictionary), the expected test results, the criteria for evaluating the tests, the steps to be taken to perform the test, and any assumptions and constraints. The final section will document the traceability of each of the test cases to the requirement(s) it addresses.

The software test plan should contain the following: a description of the scope of the tests including identification of the system and software, an overview of the system, and the relationship between this software and any other system components. This is followed by a description of the software test environment, which lists the test sites, the hardware and firmware items, the installation procedures, the tests to be performed and the participants in the testing process. The next section contains a full description of the tests to be performed. References are made to the levels of testing, the classes of test to be performed and the general test conditions. A description of the recording and analysis of the test results is also found in this part of the report. The section concludes with a list of the systems and subsystems to be tested and the tests that are to be performed on each. The scheduling of each of the tests forms the basis of the next section. It describes activities such as the tests themselves, the set-up time for the system prior to testing, the collection and/or creation of data values needed for the testing process, how the tests are conducted (including planned retesting) and the preparation and review of the software test report. The final section identifies the relationship between the tests and the software requirements identified at the beginning of the development process.

The software test report is the last of the documents produced during the testing process; this document will again outline the scope of the testing process. It also provides an overview of the test results that identifies deficiencies in the software, their impact on the rest of the software and the steps that may be taken to correct them. This section will also assess the impact of the test environment and make recommendations as to improvements that may be made. The test results section will contain a summary of the overall test results, a more detailed description of those results that did not match the expectations, and an outline of any steps that were taken to correct the problem. The test log contains a chronological listing of the dates, times, personnel and tests that were performed, together with the hardware and software combinations used and the date and time of each of the test-related activities.

Use of CASE tools

A number of general-purpose application programs can help in documenting the test data items or the testing process. Other CASE tools have been specifically developed for the testing process.

General-purpose software such as word processors can be used to provide templates for the various documents that are produced during test planning or within the testing period. For example, a word processor may be used to format a text file that has been output during a test. This will allow the test team to annotate the results in a quick, clear and consistent manner.

Other CASE tools have been specially designed to provide automated assistance in aspects as varied as test case design and analysis of output files. The following section will briefly examine the purpose and operation of some of these tools.

At the beginning of the testing process, test data items and test cases may be required. In some instances, for example a program that is to process a large number of files, a large number of data items will be needed, which may be required to conform to a particular set of rules. In this case the use of a **test data generator** may be beneficial. This tool can create a number of test cases that can be tailored to fit requirements, such as that they contain only a specified set of characters. Thus, if a postcode is to contain only four digits, the test data generator can be used to create sets of data that conform to this rule. Other data sets containing illegal characters can be created to test the program's ability to handle illegal characters during the input of a large amount of data.

An **oracle** is a program that can predict the output of a system if there are no defects in the system. An oracle cannot be completely automatic, as it would then perform the task of the program, rendering the program's development unnecessary. Thus there must be some human intervention in the operation of an oracle. A throwaway prototype may be used as an oracle since it will perform the required tasks; however, that prototype would not be any good for predictions that involved data validation (since a throwaway prototype is usually developed without regard to data validation). Human intervention would be required to predict the outcomes when test data cases are used to test the validation modules of the final software solution.

File comparators, as the name suggests, are used to automatically compare two files. This type of CASE tool is useful when a new software solution is being developed to replace an existing software solution. Provided that the format of the output files is the same, the file comparator can be used to compare files created by the existing software solution and the old one. When the comparator detects a difference in the files, it outputs a listing of the differences. These can then be analysed by the test team to detect the problems with the new solution.

CASE tools that are used to coordinate and manage software testing are known as **test management tools**. The purpose of these tools varies from conducting batch testing of programs through to performing comparisons between the expected output and the actual output of a program. Some test management tools act as generic test drivers, reading test cases from a file, formatting it for input and operating the software under test. They can also be used to test an interactive human–computer interface.

Communication with those for whom the solution was developed

Communication is one of the most important aspects of the software development process. Members of the software development team have to communicate with each other and the team also has a responsibility to keep the client informed of progress.

The testing phase is no different from any of the other stages in the software development cycle. The client has a right to be informed of the progress of the tests and of how the program measures up to the original specifications.

The reports given to the client need to be presented in a non-technical manner to allow the client to properly understand the contents. A report that hides problems by the use of highly technical language is not going to provide the information needed by the client.

The client must also have an opportunity to contribute to the evaluation of the tests that have been performed. This contribution will generally increase as the testing process progresses, but the client should be given the opportunity to contribute at all stages of the testing process.

Test results

The test results will concern the client in a number of ways. The first is that problems uncovered during testing may cause the project to be delayed. For example, the solution of a problem in the program may require the complete rewriting of a module. Also of concern to the client is the efficient use of resources. Thus, if testing shows a 10% increase in the time for processing, the client will need to be informed so that a decision can be made as to whether the increase is a problem or not. An overall picture of the actions of the software can also be gained from the test results, especially when live data has been used. This will assist management to fine tune the use of the package within the organisation.

Comparison with the original design specifications

If the program meets all of the requirements of the original design specifications, the client will be happy; if it does not, the client will expect that the program is brought up to that standard.

In order to make a judgment on whether the software meets those specifications, the client will need access to the test report or a summary of the report in less technical terms.

Exercise 6.2

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

   Testing has to be carefully __________ and executed if it is to provide the maximum amount of __________ about the software. Documentation helps with this process. The three major document types produced are a software test __________, a software test __________ and a software test __________. The software test __________ describes the __________ of the system and criteria for __________ the tests. The software test __________ contains, among other things, a description of each of the tests and the hardware, __________ and installation __________. The software test __________ provides information about the __________ of the tests and the __________ of the tests.

2 Explain why it is important to properly document the testing process.

3 A program is being created to manage a school's software collection. Explain why it is necessary for the development team to discuss the results of the test with the school's management.

4 Explain how a word-processing program can be used during the testing stage of development. Illustrate your answer with examples.

5 Create a program that will convert temperatures from Celsius to Fahrenheit and vice versa. Develop and document a set of tests for this program.

Team Activity

Your team has been told to oversee the beta testers of a new game. You need to design a form for the testers to fill in with details of the system being used to test the program and the problems that they encountered. The programming team will need the data collected from these forms so they can correct the problems. The form must not occupy more than one A4 sheet and must be easily understood by both the beta testers and the programmers.

Review exercises

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Testing and evaluation of a __________ product involves making sure that it __________ and meets the __________ specifications. Verification testing is carried out to ensure that the program performs all its __________ properly. These tests are first performed on the individual __________ using test data sets. The modules are gradually __________ into the program which is tested after each __________ module is added. Other testing procedures, often using live __________ and known as __________, aim to make sure that the program meets the design __________.

2 Explain why test data sets are created thoughtfully rather than being created at random. Use an example to illustrate your answer.

3 Design a set of test data, excluding data validation, for the following module and present it in an appropriate form.

A refrigerator alarm system is to be installed in a commercial kitchen. The alarm is to sound if the door has been left open for more than 45 seconds or if the internal temperature rises above 2°C. Inputs to the system are taken from a timer which starts when the door is opened and which stops and resets when the door is closed and from a temperature sensor. These values are converted by the system into numerical values, which represent the number of seconds and the temperature in °C, and sent to the processor.

Your choice of test data items should be chosen to perform exhaustive condition decision–condition coverage testing.

4 A computer program is being designed for the police department to process the photographs from 'speed cameras'. Create a set of test data for the processing algorithm together with the expected outputs and reasons for choosing each item of data. Write an algorithm for this problem and test it using your test data.

Code the program in one of the approved languages and use your test data to check the program. The fines are calculated according to Table 6.2.

Speed above limit	Penalty
1 to 15 km/h	Fine \$250 plus 4 points
16 to 25 km/h	Fine \$850 plus 6 points
26 to 35 km/h	Fine \$1450 plus 8 points
over 35 km/h	Fine \$2850 plus loss of licence for 2 years

Table 6.2

5 Compare and contrast acceptance testing and alpha testing as used in software development. Describe, with examples, when each of them is appropriate.

6 You have been given the task of providing live data to test a supermarket cash register system. Explain how you would choose appropriate sets of data for the testing.

7 You have volunteered to beta test a new word-processing program for *Macrohard Software.* Describe how you would approach this testing. Using your description, perform a test on the word-processing program you use at school. Report on your findings.

8 During your course you have written a number of computer programs. Choose one of them and produce a software test plan for that program. Test the program according to your plan and write a software test report.

9 Examine the application software available to you at school and describe how each of these applications could be used to help with the software testing process. If you have applications that are not suitable, list them as well, together with the reasons why they are not suitable.

10 Explain how you would communicate the results of testing to a client who has employed you as a software designer. Illustrate your answer with examples.

Chapter summary

- The two aspects of testing are validation and verification.
- Validation ensures that the software meets the design specifications.
- Verification ensures that the software functions correctly.
- Software specifications should be written to enable performance of the solution to be measured.
- There are a number of different ways to generate test data for a software solution.
- Data for statement coverage testing ensures that every statement in the module is executed at least once.
- Data for decision–condition coverage testing ensures that each decision is tested at least once in its true and false state.
- Data for multiple decision–condition coverage testing ensures that each decision is tested at least once for all possible combinations within multiple decisions.
- Data for exhaustive decision–condition coverage testing ensures that all possible combinations of true and false are tested for every decision.
- White box testing examines the detailed workings of a module by means of the tests.
- Black box testing just examines the outputs for given inputs, ignoring the processing that went on within the module.
- For black box testing, data items can be chosen for boundary analysis and equivalence partitioning.
- Boundary analysis involves choosing test data items on each side of a boundary.
- Equivalence partitioning involves choosing test data items that share particular properties. These items form a partition that is used for testing.
- Unit testing treats each of the modules as a stand-alone unit for testing purposes.
- Integration testing takes place when modules are combined to form a solution.
- Integration testing may be either top-down or bottom-up.
- After integration, the program passes through function testing with users.
- The two forms of function testing are acceptance testing for custom software and alpha and beta testing for application software.
- System testing involves testing for security, performance and recovery.
- Live test data is also used to test the solution, as it simulates the processing required when the software is installed and fully operational.
- Quality assurance methods can be used to ensure that the software is constructed to the best possible standards.
- Software testing is accompanied by documentation: a software test description, a software test plan and a software test report.
- CASE tools may be used to help in all aspects of the testing phase.
- Test results and evaluation against the original specifications are communicated to the client.

chapter 7 Maintenance of software solutions

Outcomes

A student:

- differentiates between various methods used to construct software solutions (H 1.2)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- constructs software solutions that address legal, social and ethical issues (H 3.2)
- applies appropriate development methods to solve software problems (H 4.2)
- applies a modular approach to implement well-structured software solutions and evaluates their effectiveness (H 4.3)
- applies project management techniques to maximise the productivity of the software development (H 5.1)
- creates and justifies the need for the various types of documentation required for a software solution (H 5.2)
- selects and applies appropriate software to facilitate the design and development of software solutions (H 5.3)
- assesses the relationship between the roles of people involved in the software development cycle (H 6.1)
- communicates the processes involved in a software solution to an inexperienced user (H 6.2)
- uses a collaborative approach during the software development cycle (H 6.3)
- develops effective user interfaces, in consultation with appropriate people.(H 6.4)

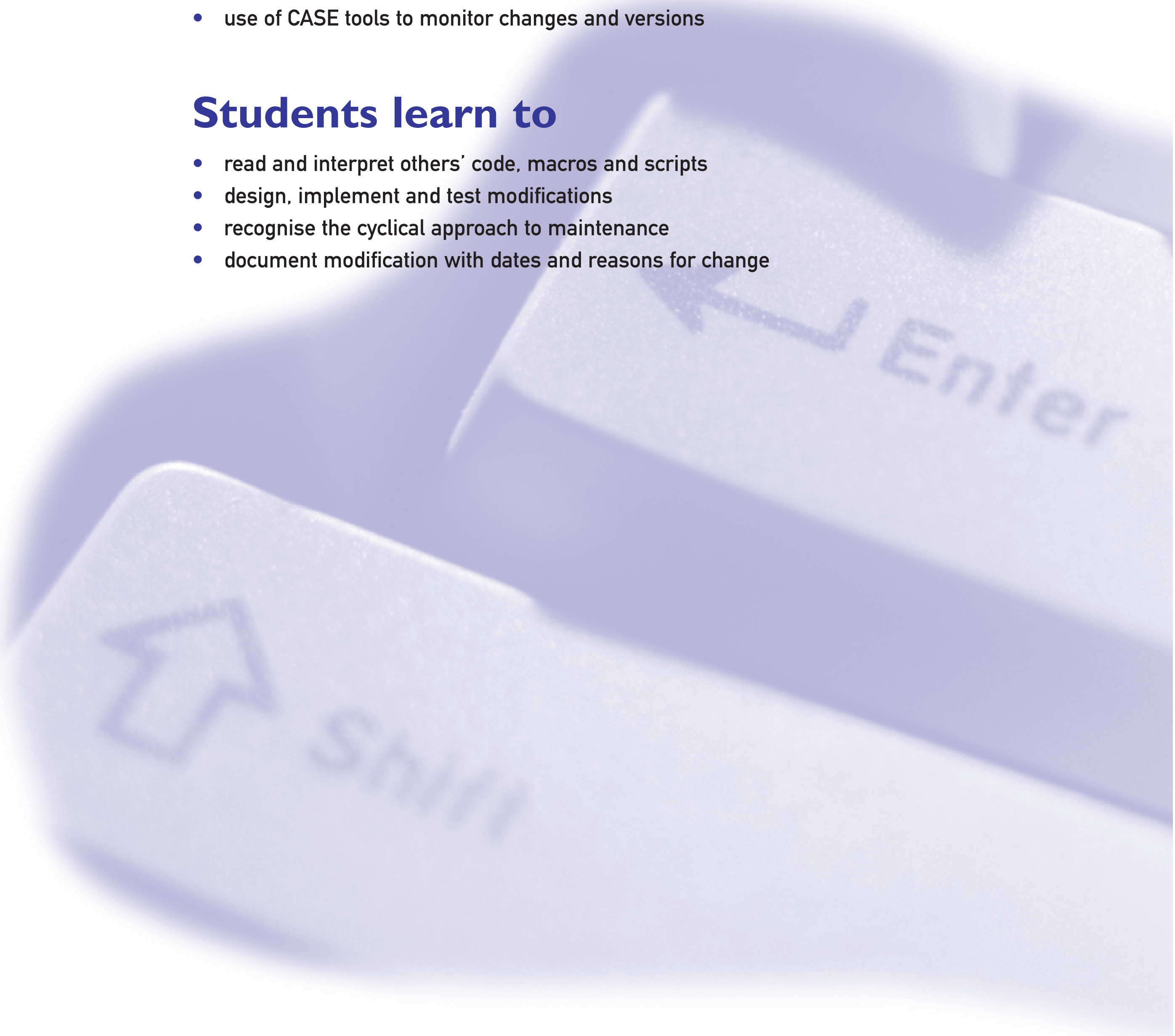
Students learn about

Modification of code to meet changed requirements
- identification of the reasons for change in code, macros and scripts
- location of section to be altered
- determining changes to be made
- implementing and testing the solution

Documentation of changes
- source code, macro and script documentation
- modification of associated hard copy documentation and online help
- use of CASE tools to monitor changes and versions

Students learn to

- read and interpret others' code, macros and scripts
- design, implement and test modifications
- recognise the cyclical approach to maintenance
- document modification with dates and reasons for change



Modification of code to meet changed requirements

The term **maintenance** is used to broadly describe any effort that is put into software after it has been implemented.

As in the Preliminary Course, any number of factors can lead to the need for maintenance. These factors include a need to remove a bug, to improve the efficiency of the program or to change the manner in which the program operates, to allow the program to cope with changes in the input data, to meet new requirements within the organisation or to comply with new government requirements. Maintenance may also be necessary when changes are made to the hardware or to the software such as the operating system.

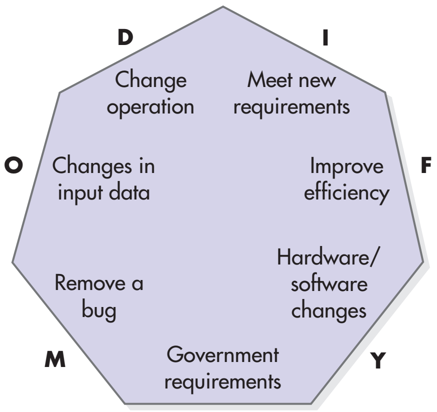


Figure 7.1 Maintenance may be required for a number of reasons.

Changing user requirements

As a user becomes more familiar with and competent in using a computer program, shortcomings in its operation may become evident. These shortcomings may be in the way that the program works or, more often, because the user wishes or needs to perform a further task.

Upgrading the user interface

When software is originally developed, the user interface receives a lot of attention. However, even though this effort is put into development, the user interface often contains a number of aspects that would benefit from redesign.

Redesign of the user interface may be needed because the user, in becoming more familiar with the original interface, has discovered design problems. It is quite common for this to happen as, although the user is involved with the original interface design, its long-term use may bring to light problems that had not been thought of or have been brought on by slightly different work practices. This can be especially true in the design of menus and key combinations used for shortcuts.

Changes in the data to be processed

Changes in the form of data to be processed will often lead to a need to modify software. The most recent world-wide example of this problem was in the correction of the so-called 'Millennium bug' in which the two-digit representation of the year in a date, used in the last half of the 1900s and identified as a potential source of problems, needed to be changed to four digits. Date-dependent software had to be modified in order to accommodate the new representation of the year.

Expansion of the existing system to allow for a greater number of data items than was originally envisaged when the software was designed may involve revising the representation of data items and/or data structures. These revisions then have to be incorporated into the software if it is to function properly. An example of this in Australia's recent history was the conversion of all telephone numbers to eight digits in the late 1990s in order to increase the number of possible telephone connections.

Another source of change in the data to be processed is brought about by the changing needs of the organisation. In this case new data items and processes may need to be incorporated into the software application in order to obtain the required outputs. For example, when government legislation was introduced requiring all citizens to have a tax file number, the banks had to modify their software so that it could store and retrieve customers' tax file numbers as well as use the tax file numbers in the calculation of interest.

Figure 7.2 Introduction of new technology may force an organisation to modify its software.

Introduction of new hardware or software

Custom software generally has a long life within an organisation as it is initially expensive to produce. Cases of software lives of ten to fifteen years are not uncommon. However, during the life of the software, technology is improving both the hardware and the operating system software. For software to last as long as it does, maintenance will be required to allow it to run with the new technology and to take advantage of the benefits that the new technology offers.

Changing organisational focus

An organisation is not static. Its goals and purpose move through a continual process of evolution. As the organisation evolves, its focus changes. For example, P&O, which started as a shipping company, with its focus on providing a service to Asia and Australasia. has now evolved into a company that owns cruise liners and holiday resorts, provides freight services, runs a cleaning service and operates wharves.

As the focus of an organisation moves through change, the software needs also change. Companies will generally move into areas that are related to their original purpose, and so the new software needs will generally be similar to the previous ones. The current software will usually be able to perform a majority of the required tasks, so it is obviously more economical in both time and cost to modify the current software than to create a completely new package.

Changes in government requirements

Government regulations and laws are continually changing to reflect social, political and economic circumstances. As the laws change, it may be necessary for an organisation to modify its software in order to comply with the new requirements.

Areas in which these changes are most evident to organisations are those that involve the payment of taxes. However, there are other instances in which software may need modification. For example, the government might require a company to track the use of particular chemicals or to furnish details of the purchasers of their products. In these cases existing software will need modification in order to comply with the government's wishes.

Poorly implemented code

Software developed many years ago may not have been as structured as it is now. The emphasis sometimes was on the software being able to perform its task no matter what the cost. As the software comes up for maintenance, evidence may arise that shows that the current implementation is not as efficient as it could be, or that a software patch has been used to work around a problem. Maintenance is a good time to evaluate the code for its efficiency and to properly overcome earlier problems rather than rely on a software patch.

Identification of the reasons for change in code, macros and scripts

Maintenance is not about creating a 'quick fix' for the identified problem but about creating a lasting solution. To this end, maintenance should follow the same steps as any other problem-solving activity. That is, define the problem, design a solution, test the solution and implement it.

The first step in maintenance is to determine the needs of the user that are not being currently provided by the software solution. As the users become more adept at using the software, they may see the possibility of using the system in a new way or they may seek improvements to the manner in which a task is performed. For example, the users might find that the human interface of the software is not as intuitive as the program design team had anticipated. The users' suggestions as to how they would like this interface to work would be passed along to the maintenance team for consideration and would be used as a basis for the modification design specifications.

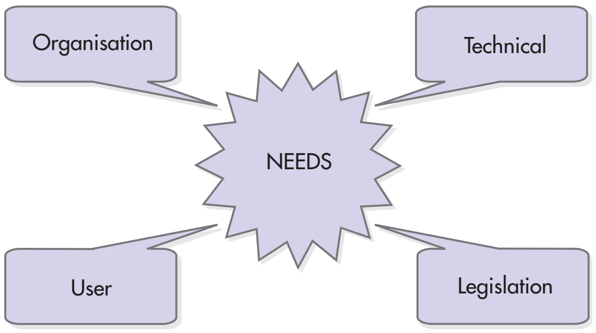


Figure 7.3 Needs determine the changes that are to be made to software.

Sources external to the user, such as management or government legislation may have needs that are not being met. In these cases, a set of requirements for the modified software is necessary, and these are constructed from a comparison between the tasks currently being performed by the software and those needed to comply with the new needs. The requirements may be as simple as complying with a new scale of tax deductions, or they may involve complex processes such as reorganising the company's computer program to comply with new bookkeeping regulations.

The easiest maintenance needs to identify are those associated with faults in the software. Even though it is tempting just to 'fix' the problem, the requirements should also be documented. Full documentation of a software solution is very important for subsequent maintenance, as it tells the maintainers exactly what has been achieved up to that moment.

As time passes, technology becomes outdated and is eventually unavailable. A software solution may not function correctly when used with new hardware or a newer operating system. Again, it is important to carefully document the requirements for maintenance. As with the correction of faults, proper documentation is essential for future maintenance.

Location of section to be altered

Once the new requirements have been established, the maintenance team has the task of changing the software so that it meets the new requirements. Some of the new requirements can be met by the addition of new modules, whereas other requirements will be fulfilled by the modification of existing code.

Development of new modules is generally a more straightforward process than modification of existing ones. The main reason for this is that the new modules contain no pre-existing structure, and, provided they interface correctly with the rest of the program and perform their tasks, the program should work properly. Modification of a module that already exists requires the team to understand the workings of that module, be able to locate the section that needs changing, and incorporate the changed section into the module. A modular approach to software development will assist the team to locate the section to be changed. If a program has been written as a whole, without a modular structure, location of a section that has to be changed is a very difficult task.

In some cases a whole module may need replacement. These sections should be fairly easy to locate in a well-documented solution, as the documentation will lead the maintenance team to the appropriate parts of the program. (The initial use of appropriate names for modules is a most important aid.) A whole module would need to be replaced, for example, when a new input device, such as a scanner, did not present data to the program in the same form as the device it was replacing, perhaps a keypad. A module that accepts the data in the new form would need to be written to replace the input module of the program.

A harder task is to locate small sections of code that need to be replaced. In well-documented code, the internal documentation can be used to find the section that needs replacement. The 'find' function of a text editor can be used to help with this aspect. If the program development environment does not contain this function, the text could be exported to a text processor that does support it.

Sometimes the value of a constant needs to be changed. This can cause problems if the constant value has not been assigned an identifying name at the beginning of the program or module. When a constant has been defined at the beginning of a section of code, it is easy to find and, more importantly, we know that the change in value will be carried right through the code. If the constant has been used as a numerical value in the code section, each occurrence needs to be located, with the possibility that one or more will be missed. The following example uses a simple program to illustrate this point.

In Sample 1 the value will need to be changed four times, whereas only one change is needed in Sample 2.

Example 1

A program has been written to calculate the areas and volumes of a number of figures and solids that contain circles. Greater accuracy is required after the program has been written. The program team has been asked to change the value of pi from 3.14 to 3.14159. In the first section of the BASIC program, the value of pi has been used in each of the formulae; the second code section defines a constant, PI, at the beginning of the code and this identifier is used throughout the program.

Sample 1

```
10  PRINT 'SAMPLE PROGRAM USING PI'
20  PRINT 'PLEASE INPUT A RADIUS'
30  INPUT RADIUS
40  CIRCUMFERENCE = 2 * 3.14 * RADIUS
50  AREA = 3.14 * RADIUS * RADIUS
60  SURFACE AREA = 4 * 3.14 * RADIUS * RADIUS
70  VOLUME = 4 * 3.14 * RADIUS * RADIUS * RADIUS
80  PRINT 'THE CIRCUMFERENCE OF THE CIRCLE IS'; CIRCUMFERENCE
90  PRINT 'THE AREA OF THE CIRCLE IS'; AREA
100 PRINT 'THE SURFACE AREA OF THE SPHERE IS'; SURFACE AREA
110 PRINT 'THE VOLUME OF THE SPHERE IS'; VOLUME
120 END
```

Sample 2

```
10  PRINT 'SAMPLE PROGRAM USING PI'
15  PI = 3.14
20  PRINT 'PLEASE INPUT A RADIUS'
30  INPUT RADIUS
40  CIRCUMFERENCE = 2 * PI * RADIUS
50  AREA = PI * RADIUS * RADIUS
60  SURFACE AREA = 4 * PI * RADIUS * RADIUS
70  VOLUME = 4 * PI * RADIUS * RADIUS * RADIUS
80  PRINT 'THE CIRCUMFERENCE OF THE CIRCLE IS'; CIRCUMFERENCE
90  PRINT 'THE AREA OF THE CIRCLE IS'; AREA
100 PRINT 'THE SURFACE AREA OF THE SPHERE IS'; SURFACE AREA
110 PRINT 'THE VOLUME OF THE SPHERE IS'; VOLUME
120 END
```

Determining changes to be made

The new set of requirements is used to determine the nature of the changes to be made to the program. As seen, some of these changes will involve the writing of one or more completely new modules, whereas others will involve changes to the logic or minor adjustments such as the use of a new value for a constant.

The reasons for the changes need to be considered before attempting to find the section that has to be changed. If the required change is due to an error in the logic of the program, the source or sources of the problem have to be identified. These sources can usually be determined from the test documentation and the original algorithm descriptions and source code. Changes made under these circumstances will involve rewriting sections of code to eliminate the problem.

Changes in the user interface often involve minor adjustments to various screen elements. If one or more screens require major modification, if is often best to start from scratch and design a completely new layout. If this is necessary, the screen must be designed within the specifications already used for the other screens.

The addition of new functions to an existing program is by far the most difficult task faced by a programmer. The new section needs to be added without affecting the operation of the existing program. Thus the new module(s) have to be able to pass parameters to and receive parameters from the existing program without interfering with its operation.

Implementing and testing a solution

Once changes have been made to a program, the maintenance team needs to implement and properly test the changed sections of code and the changed program.

Implementation of the changes will normally be made in the same development environment as the original application. Thus, if a program has been developed using TUSIL, the modifications will be made in TUSIL, as changes can be made to the old code or new code can be added to the old.

The original test data sets can be used to check the operation of modified modules. However, this practice may lead to errors passing through the test process if slight changes have been made to the specifications. It is a good practice to check all test data items against the revised program specifications before allowing their use in the testing process.

The rules of testing and evaluation should not be discarded during modification, as any lapses in documentation of the test process itself may have serious consequences when the application comes up for review and modification at a later date.

If changes have been made to the user interface as a result of feedback from the user, it is appropriate for the users to be involved in the evaluation of the new interface before it is implemented.

Exercise 7.1

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Maintenance is a broad term used to describe any ________ that is put into software after ________. Factors that can lead to maintenance include a need to ________ a bug, improve the ________ of the program, change the way in which the program ________, meet new ________ of the ________ or comply with new ________ regulations. Changes in ________ or ________ may also lead to a need for ________ in the software.

2 Describe how you could determine the changes to a software system needed by an organisation.

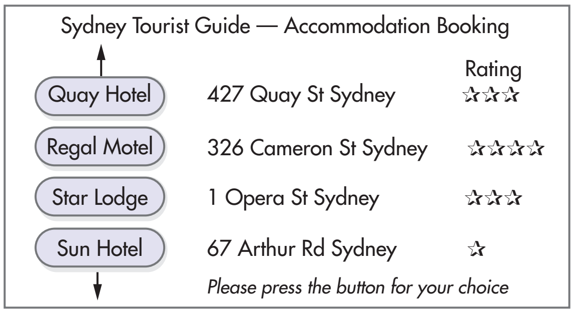
3 Explain, in your own words, the factors that may lead to a need to change a software system.

4 You are in charge of the software maintenance team of a large organisation. Describe how you would locate the sections of the program that will need to be changed during maintenance. Write a memo for the other team members, describing the steps you want them to take in locating the sections of code to be changed.

5 Examine the operation and user interface of a software application in use at your school, for example a library enquiry system. For this system, report on the ease of use and make recommendations for any changes.

6 During this course you have written a number of programs. Choose one of these programs and have a friend use it. From the comments made by your friend, determine the changes that can be made to the program. Implement the changes and properly document them.

7 Figure 7.4 shows a screen from an interactive tourist guide. The Tourist Information Board has asked you to redesign the screen to make it more user friendly. One of the problems noted by the Tourist Information Board is the lack of any navigation tools on the screen.


Sydney Tourist Guide — Accommodation Booking

		Rating
Quay Hotel	427 Quay St Sydney	☆☆☆
Regal Motel	326 Cameron St Sydney	☆☆☆☆
Star Lodge	1 Opera St Sydney	☆☆☆
Sun Hotel	67 Arthur Rd Sydney	☆

Please press the button for your choice

Figure 7.4

8 Explain why it is important to properly test software after performing any modifications.

9 Describe the types of test that should be performed after software has been modified. Explain the purpose of each type of test.

Documentation of changes

As seen, documentation forms a very important part of the software development process. This is especially important during the maintenance stage when different versions of the same software may be in use. It is a very difficult task to keep track of the changes that have been made and the software versions that are in use in an organisation. For example, a new version of the software may be in use in one department on a trial basis, while the rest of the organisation is still using the earlier version. The process of keeping track of software versions and their documentation is part of the management processes associated with a large software installation. Management of software resources is known as **configuration management** or **CM**.

Configuration management consists of a number of processes: configuration planning, control and management of system change, system building, and the management of versions and releases of software.

Configuration planning is the process whereby the documents essential for further development are identified and placed under configuration control. Documents that fall into this category include specifications, designs, algorithms and code. A further task undertaken in configuration planning is to assign a unique name to each of the documents so that they may be identified later on. It might appear that configuration planning is an isolated activity that is kept for one point in the development process, but it actually takes place during the whole of the development activity.

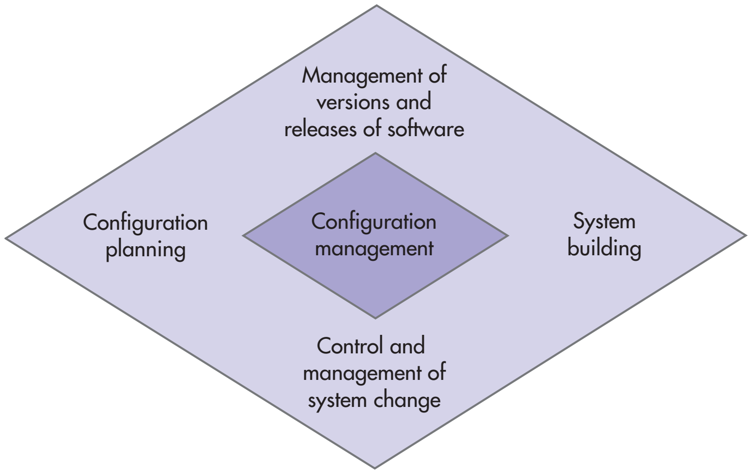

Management of versions and releases of software
Configuration planning
Configuration management
System building
Control and management of system change

Figure 7.5 Configuration management consists of a number of processes that are used to control the changes in a software system.

Change control is concerned with ensuring that any changes made to the software are done in a controlled and predictable way. Without any form of control, conflicting changes could be made by different maintenance teams.

System building involves acquiring the particular software components and assembling them into a system that meets the required goals. During maintenance, a large number of these software components will be provided by the pre-existing software system. Others will need to be written or obtained from another source. System building may also require that existing components be modified to meet the needs of the user.

The management of versions and releases requires that a suitable system of naming be devised to distinguish between the different versions and releases of the software. This is most important if there are to be a number of different releases of the basic software package to meet the requirements of users in different situations. One of the responsibilities of the CM team is to determine when a new release or version is to be created.

Source code, macro and script documentation

An important part of the development documentation is the source code and its associated documentation. The term 'source code' is used in its broadest sense to describe any code that is modifiable, including macros and scripts.

Documentation of the source code should follow the rules already discussed in this course. That is, the code should contain as much intrinsic documentation as possible (for example the choice of appropriate identifiers and the use of modularisation) as well as notes and comments. The code should be written in such a way as to expose the structure of the process. Descriptions of the process and any accompanying design documentation will also assist in the maintenance process.

Modification of associated hard copy documentation and online help

Changes to software cannot be made without making the necessary changes to documentation that accompanies the product. The user is entitled to documentation that refers to the current version of the software being used. This means that the changes made to the software during maintenance have to be quickly reflected by changes to the documentation.

Maintenance of hard copy documentation presents a greater problem than online documentation. Online documentation is generally updated at the same time as the changes are made to the software and distributed electronically over a network or on a medium such as a CD. With hard copy documentation, the changes have to be made to the hard copy which is then printed and distributed to the user. The user then has to ensure that the new documentation is put in the appropriate place in the pre-existing documentation. This process also puts the onus on the user to remove any outdated sections of the documentation.

It is more prevalent now for hard copy documents to be published in a portable document format that allows users to read the hard copy online or print the relevant sections for their use. This ensures that each user has access to the most recent version of the document.

Use of CASE tools to monitor changes and versions

A number of configuration management tools exist to assist in the process of configuration management. These tools provide for the CM team to keep track of version histories, documentation and change by means of integrated databases combined with a means of document production. CASE tools can also be used to assist in system building by being able to combine various components into a software solution. Some CASE tools are even able to specify what components are needed and how they can be combined to create a working solution.

Application software can also be used in the management of software. Word processors are used in documentation, databases are used to help manage versions, management software is used to create schedules and manage the resources needed for maintenance, and graphics packages are used to assist in providing support for the user.

Exercise 7.2

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Management of software changes is known as __________ management. __________ management consists of four different processes: __________ planning, control and __________ of system __________, system __________ and the __________ of versions and __________ of software. All these processes rely on __________ documents being identified and placed under __________ control.

2 Explain how the process of configuration management helps with the documentation of changes to a software system.

3 List the documents that are needed during maintenance. Briefly explain how each of these documents is used during program modification.

4 Investigate and report on the differences between a version of a program and a release of a program.

5 Describe the ways in which a programmer can help the process of maintenance when first creating a software product. Give some examples to support your answer.

6 Describe the ways in which CASE tools can be used to help with the process of configuration management.

7 List the software applications installed on your school computers that can be used to help with configuration management and the tasks that each of the applications will perform in configuration management.

8 Describe the types of change that have to be made to user documentation associated with software that has been modified. Describe the ways in which these changes can be passed on to the user.

Review exercises

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

Software maintenance is any process that ________ software ________ installation. Maintenance should be directed by the ________ management team so that it proceeds in a ________ manner. The ________ of the organisation and the ________ determine the changes that are to be made to the software. These ________ are conveyed to the ________ management team who decide when the ________ should be made.

2 Describe the types of needs of an organisation that can lead to software being modified.

3 Explain how proper documentation can help with the process of software modification.

4 Choose a software application you have written and determine suitable changes to it. Write a set of specifications for the modifications.

5 Update the user documentation for the program you chose for question 4 to reflect the changes that you would make to the program.

6 Compare two versions of a commercial application program such as a word processor. List the differences between the versions including enhancements that may have been made to allow it to operate more efficiently with updated hardware or operating systems.

Team Activity

Devise an appropriate naming method for the documents produced during software development. Your naming system should allow documents to be added during maintenance. Choose a program that you have written and name the documentation for that application using your naming method.

Chapter summary

- Maintenance is a term used to describe any effort put into software after it has been implemented.
- A user may, through experience in using the software, find shortcomings in its operation.
- The user interface may need upgrading to overcome design problems.
- Software may need to be modified to cope with the changed format of input data.
- Expansion of the system may lead to a need for modification of the software.
- Changing needs of the organisation can lead to a need to modify the software.
- Changes in government legislation can lead to a need to change existing software.
- The introduction of new hardware and/or software can bring about a need to change the existing software.
- Poorly implemented code often has to be modified if the program is to work efficiently.
- The reasons for change must be clearly identified and documented.
- Before changes can be made, the appropriate section of code has to be identified.
- Some changes are as simple as modifying one or two small sections of code; others require that new modules be written.
- The changes that are made as a result of new needs should be properly documented, as in the rest of the software development process.
- Changed software should be thoroughly tested before implementation using the same testing strategies as are applied to new software.
- The management of software resources is known as configuration management.
- Configuration management consists of the planning of configurations, control and management of all system changes, management of the building of the system and management of the versions and releases of the software.
- Configuration planning identifies and catalogues those documents essential for further development of the software.
- Change control is a process whereby all changes to the software are done in a logical and controlled manner.
- System building is the process of obtaining the necessary software components and assembling them into a system that meets the organisation's requirements.
- Version and release management is concerned with the orderly release of the different variations of the software within the system.
- All source code, macros and scripts should contain and be accompanied by the appropriate documentation.
- Hard copy and online documentation should be updated to reflect the changes made to the software system.
- A number of CASE tools have been designed to assist with the process of configuration management.

chapter 8
Developing a solution package

Outcomes

A student:

- explains the interrelationship between hardware and software (H 1.1)
- differentiates between various methods used to construct software solutions (H 1.2)
- describes how the major components of a computer system store and manipulate data (H 1.3)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- constructs software solutions that address legal, social and ethical issues (H 3.2)
- identifies needs to which software solutions are appropriate (H 4.1)
- applies appropriate development methods to solve software problems (H 4.2)
- applies a modular approach to implement well-structured software solutions and evaluates their effectiveness (H 4.3)
- applies project management techniques to maximise the productivity of the software development (H 5.1)
- creates and justifies the need for the various types of documentation required for a software solution (H 5.2)
- selects and applies appropriate software to facilitate the design and development of software solutions (H 5.3)
- assesses the relationship between the roles of people involved in the software development cycle (H 6.1)
- communicates the processes involved in a software solution to an inexperienced user (H 6.2)
- uses a collaborative approach during the software development cycle (H 6.3)
- develops effective user interfaces, in consultation with appropriate people (H 6.4)

Students learn about:

Defining the problem and its solution

- defining the problem
  - identification of the problem
  - idea generation
  - communication with others involved in the proposed system
- understanding
  - interface design
  - communication with others involved in the proposed system
  - representing the system using diagrams
  - selection of appropriate data structures
  - applying project management techniques
  - consideration of all social and ethical issues
- planning and design
  - interface design
  - selection of software environment
  - identification of appropriate hardware
  - selection of appropriate data structures
  - production of data dictionary
  - definition of required validation processes
  - definition of files—record layout and creation
  - algorithm design
  - inclusion of standard or common routines
  - use of software to document design
  - identification of appropriate test data
  - enabling and incorporating feedback from users at regular intervals
  - consideration of all social and ethical issues
  - applying project management techniques

Systems implementation

- implementation
  - production and maintenance of data dictionary
  - inclusion of standard or common routines
  - use of software to document design
  - translating the solution into code
  - creating online help
  - program testing
  - reporting on the status of the system at regular intervals
  - applying project management techniques
  - enabling and incorporating feedback from users at regular intervals
  - completing all user documentation for the project
  - consideration of all social and ethical issues
- maintenance
  - modifying the project to ensure an improved solution

Students learn to:

- define the problem and investigate alternative approaches to a software solution
- select an appropriate solution
- produce an initial Gantt chart
- use a logbook to document the progress of their project
- document the software solution
- generate a fully documented design for their project after communication with other potential users
- implement a fully tested and documented software solution in a methodical manner
- use project management techniques to ensure that the software solution is implemented in an appropriate time frame
- communicate effectively with potential users at all stages of the project to ensure that it meets their requirements
- ensure that relevant ethical and social issues are addressed appropriately

Developing a solution package

This chapter examines closely the steps required to develop a solution package. We will work our way through a case study, defining and understanding a problem, going through the planning and design stages and, finally, implementing the solution. The chapter builds on the content dealt with in other chapters of the text. It is suggested that you refer to the relevant chapters as you work through the case study.

Case study

Your client has recently purchased a stationery/office supplies business. The business has been thriving and continues to do steady business. However, the previous owner refused to make use of computing technologies in the running of the business. His sole concession to technology was the use of an old calculator and a very basic electronic typewriter. He also refused to stock or sell computer consumables such as floppy disks, recordable CDs, software, printers and printer consumables such as ink cartridges. All paperwork for the business has been done by hand and all stock and customer records are kept on cards. An external accounting firm processes all accounts. Twice a month invoice slips are sent to the accounting company. The accounting company then compiles all sales for each customer into a single invoice at the end of each month. These invoices, together with preprinted mailing labels, are then returned to the shop where they are folded and placed into envelopes for posting. Orders and invoices are processed by hand. Many customers pay by direct deposit to the company's account; however, the previous owner relied on printed statements from his bank, with a three-week time lag causing customers to receive reminder notices for payments they had already made. He would not avail himself of Internet banking facilities offered by his bank and there is no EFTPOS or electronic credit card facility in the shop.

Your client wishes to computerise the business. He wants to transfer all stock and customer details into a database. He has plans to offer online purchasing and also wishes to install a small network on the premises that can later be used for training purposes. Your client also wishes to carry products that have not been carried by the previous owner and will need to add these to his stock database. He wishes to be able to automate the system as much as possible so that he can concentrate on establishing himself as an Internet service provider and the training services area of the business. The previous owner, his son in-law and two female employees currently run the business. In order to keep overheads low, the new owner and his wife will run the business with the possibility of retaining the two female employees and the previous owner's son in-law for three days a week. The previous owner and his son in-law will facilitate the handover of the business by working with the new owner for a period of two weeks.

Applying project management techniques

As seen in the Preliminary Course, a project is a specific goal or objective that needs to be accomplished in a finite time with finite resources. The case study outlined is just that. Its goal is that, at the end of a finite amount of time, the client's new business will be computerised. This has to be completed within a specified budget.

At first the task seems daunting. A careful reading of the requirements reveals a complex set of interrelated tasks. One of the initial stages of project management is to identify the various tasks involved and to break the project up accordingly. These tasks are then assigned to the appropriate team members.

When developing any system solution, a project team is usually put into place. This team consists of people who have expertise in a variety of areas and is lead by a project manager who has responsibility for the project as a whole. When putting together the project team, people are often hired from outside the company on a contractual basis if existing employees do not hold the required set of skills. For example, a software solution may require expertise in a particular programming language that is not one of the mainstream languages. Some of the work may even be contracted out to other companies, for example companies that specialise in creating code. This is called outsourcing. Outsourcing needs to be handled very carefully (see Figure 8.1).

This chapter examines the steps necessary to produce a solution to the problem outlined in the case study. The scope of the case study could be the topic of an entire text on its own. The development of the database part of the solution will be covered here in detail; the other elements of the solution will be commented on where appropriate in order to maintain the context of the database solution. We will work through the solution as though we were one of a group of teams working on this solution. Our responsibility will be to develop the databases and create a web interface to them. As a team within the project team, we will need to liaise with other teams, our project manager and at times with the client directly.

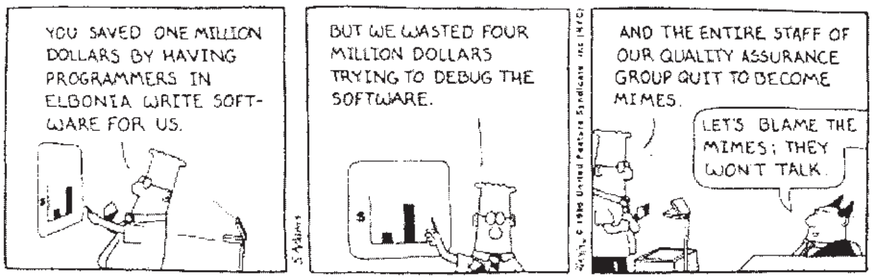

YOU SAVED ONE MILLION DOLLARS BY HAVING PROGRAMMERS IN ELBONIA WRITE SOFTWARE FOR US.
BUT WE WASTED FOUR MILLION DOLLARS TRYING TO DEBUG THE SOFTWARE.
AND THE ENTIRE STAFF OF OUR QUALITY ASSURANCE GROUP QUIT TO BECOME MIMES.
LET'S BLAME THE MIMES; THEY WON'T TALK.

Figure 8.1 Outsourcing needs to be done with great caution.

Exercise 8.1

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.
Project management involves the creation of a __________. This team includes a __________ and a variety of people with skills in desired areas. The __________ has responsibility for overseeing the entire project. It is the role of the project team to liaise with the __________ and to gather as much __________ about the client's requirements as possible. Quite often various tasks are __________; that is, they are given to contractors outside the company working on the project. This is often done when the company does not have any employees with a certain set of required __________.

2 Consider the nature of software development and project management. Discuss the importance of having a multidisciplinary project team on any given software development project. In your discussion address the pros and cons of outsourcing.

3 Form a project team for the given case study. You will need to have a minimum of four people in your team. Assign a role to each team member, for example project manager and so on. Hold a preliminary planning session in which tasks and long-term goals are allocated for each team member. Hint: If you find that your team does not have all of the required skills to complete the system, use outsourcing.

4 Draw an initial Gantt chart for the case study. You do not need to go into great detail at this stage. Make sure that the Gantt chart shows the major milestones of the project. Your timeline for the project will be eight weeks from start to finish. Make sure that your chart takes into account all aspects of the project.

5 Create a form that will help you to gather preliminary information from your client. This form will need to have areas for all aspects of the project. Use the case study description above to help you develop the form.

Defining the problem and its solution

Defining the problem and its solution consists of three main tasks: defining the problem, understanding the problem and planning and designing the solution. Each of these tasks itself consists of a range of sub-tasks. These tasks will be looked at in relation to the case study. If the crucial step of defining the problem is missed out, the entire project will suffer.

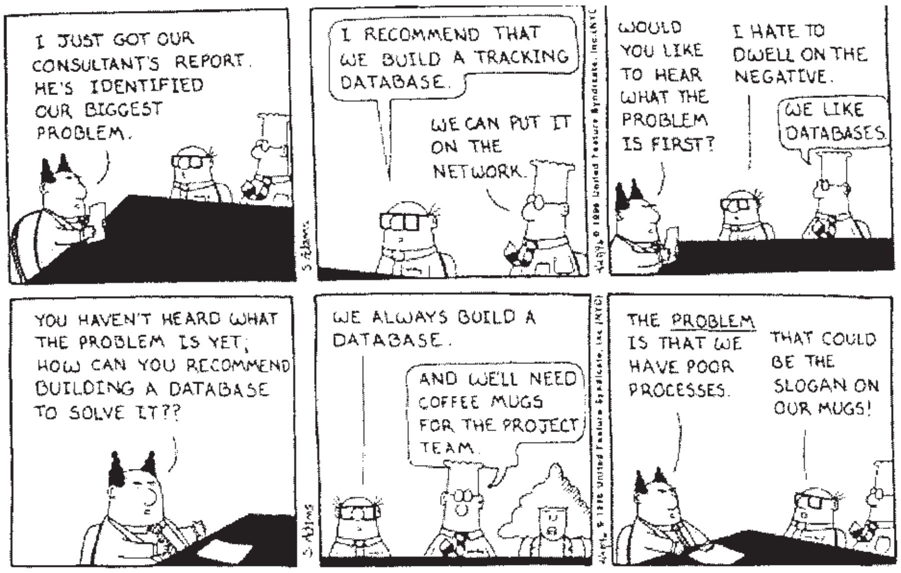

I JUST GOT OUR CONSULTANT'S REPORT. HE'S IDENTIFIED OUR BIGGEST PROBLEM.
I RECOMMEND THAT WE BUILD A TRACKING DATABASE.
WE CAN PUT IT ON THE NETWORK.
WOULD YOU LIKE TO HEAR WHAT THE PROBLEM IS FIRST?
I HATE TO DWELL ON THE NEGATIVE.
WE LIKE DATABASES.
YOU HAVEN'T HEARD WHAT THE PROBLEM IS YET; HOW CAN YOU RECOMMEND BUILDING A DATABASE TO SOLVE IT??
WE ALWAYS BUILD A DATABASE.
AND WE'LL NEED COFFEE MUGS FOR THE PROJECT TEAM.
THE PROBLEM IS THAT WE HAVE POOR PROCESSES.
THAT COULD BE THE SLOGAN ON OUR MUGS!

Figure 8.2 Problem identification is crucial.

Defining the problem

Defining the problem typically consists of three stages: identification of the problem, ideas generation, and communication with others involved in the proposed system. It is important to remember that these stages are not always clearly separated from each other and are not always completed in this order. There is overlap and quite often ideas generation is started well before the problem is completely defined. Also, identification of the problem is often done in consultation with the client and this can be seen as being a part of the third stage (communication with others involved in the proposed system).

Identification of the problem

This first stage involves looking closely at the client requirements and the existing system.

From the case study we see that the current system is completely manual; that is, there is no use of computers or computer-based technologies to aid in the running and tracking of accounts and stock levels. All transactions are processed by hand and this creates a variety of problems. For example, there is the generation of reminder notices to customers who have already paid their bills by direct deposit to the company's bank account. The system also requires manual updating of the stock database. All stocktaking tasks are also done manually and are therefore quite time consuming. There is no facility for customers to check the availability of products other than by visiting the shop or phoning in; these are time-consuming tasks for both the customer and the business. Phone orders are handwritten and in many cases duplicates are made. One copy is for filing and the second is used to fill and process the order. Depending on the nature of the order and stock availability, this can lead to the generation of further paperwork in triplicate.

Your client wishes to streamline the business. He has the following requirements:

- The stock database is to be updated automatically whenever sales are made and new stock is purchased.

- Stocktake reports are to be generated at regular intervals.
- There needs to be a facility to track customer purchases to identify purchasing trends. That is, if customer X buys a certain quantity of a product at regular intervals, stock can be ordered to coincide with these intervals, thus reducing the time stock has to be kept on hand.
- Products not previously sold by the business are to be added to the database.
- A website with online purchasing facility is to be managed in-house and not by an Internet service provider (ISP).
- The existing card database of stock and customer records is to be computerised.
- Accounts are to be handled in-house; that is, they will no longer be done by the external accounting firm.
- A small local area network, a server and five workstations are needed.
- A facility is needed to calculate discounts automatically for individual customers based on a variety of factors such as quantity ordered, type of customer, or historical factors (e.g. this customer always receives a 10% discount; this one always receives a 5% discount).

You will notice from the list that the client has specified relatively generic wishes. It is up to you and your project team to analyse these requirements in order to work them into the most suitable solution for your client. You will need to examine this list carefully to see where the solution to one requirement can be used to solve a number of the problems presented. For example, the database that is to be created for the business can be used to generate orders, invoices, stock reports and so on. It can also be linked to the website, thus solving a number of problems with a single solution.

Simply stated, the problem presented is to convert the current manual system to a computerised one, and in the process to add to the system those things that your client has identified as having been lacking to date.

Ideas generation

The next stage of the development cycle is to analyse the problem more closely and to generate ideas as to possible solutions. At this stage a number of solutions need to be put forward for consideration. These are then worked into a feasibility study that is presented to the client. The feasibility study will outline the current system and at least three possible solutions. The feasibility study should identify the pros and cons of each solution and, finally, identify a single solution as the recommended one.

Having examined the client's list of requirements, it is apparent that the best starting point for a solution to the problems is the creation of the small local area network (LAN), as this is where the databases will be housed and this is also the most logical place for the website to be housed since the client wants it to be done in-house.

The client has experience with Windows/Intel-based computing systems and has requested that any solutions be based on this platform. There are currently three alternatives for setting up a LAN. Apart from the many personal networking systems and other proprietary systems, these are Novell, Windows NT and Linux. There are advantages and disadvantages in using each of these, which need to be researched in order to give the client the best chance of making an informed choice.

The LAN is only the beginning of the solution. For the LAN to be of any use, there needs to be considerable thought put into the inputs, processes and outputs that will be involved in the system. Careful examination of the client's requirements reveals a need for some specialised software. The client will need an accounting package, a database management system and web-serving software. There is also a need for word-processing software and perhaps a spreadsheet facility in order to model business strategies and so on. The client has expressed an interest in the use of MYOB (Mind Your Own Business) for account-keeping purposes. The product catalogue from the supply company is available on CD-ROM as an Access database. Word-processing and desktop-publishing software can also be used for correspondence and for the creation of flyers for advertising.

Communication with others involved in the proposed system

An indispensable element of any solution package development is communication with others involved in the proposed system. This is a recurring element at many stages of the development process and is essential if the final product is going to be useful to the client. In the initial stages of the development process, it is common for the client and the project team to have regular meetings and conversations. These will occur before even one line of coding is attempted. It is very costly to rewrite code or to throw out code and start again. Time spent in consultation with the client and as many users of the new system as possible will make the later stages much more productive and cost effective.

In an initial meeting with the client, various notes were made, including a rough diagrammatic representation of the existing system. Information was gathered on how stock is ordered, how orders are processed and how accounts are organised. The following has been adapted from the initial handwritten notes.

Stock for shop (this may be to replenish supplies or to fill an order) Note: The current system uses sales tax but this will need to be changed to accommodate the changeover to the GST (Goods and Services Tax).

1. *Stock arrives*
2. *Stock is recorded on cards*
3. *Locate card*
4. *Record quantity in, quantity out (if stock for specific order), quantity remaining in stock*
5. *Record cost price*
6. *Add sales tax (22%)*
7. *Add mark up (3% to 100% depending on a number of factors)*
8. *Remove sales tax for ex tax clients*
9. *Tick item off invoice*
10. *Price tag each item*
11. *Locate storage area and store item*
12. *Re-price old stock with new price if applicable*

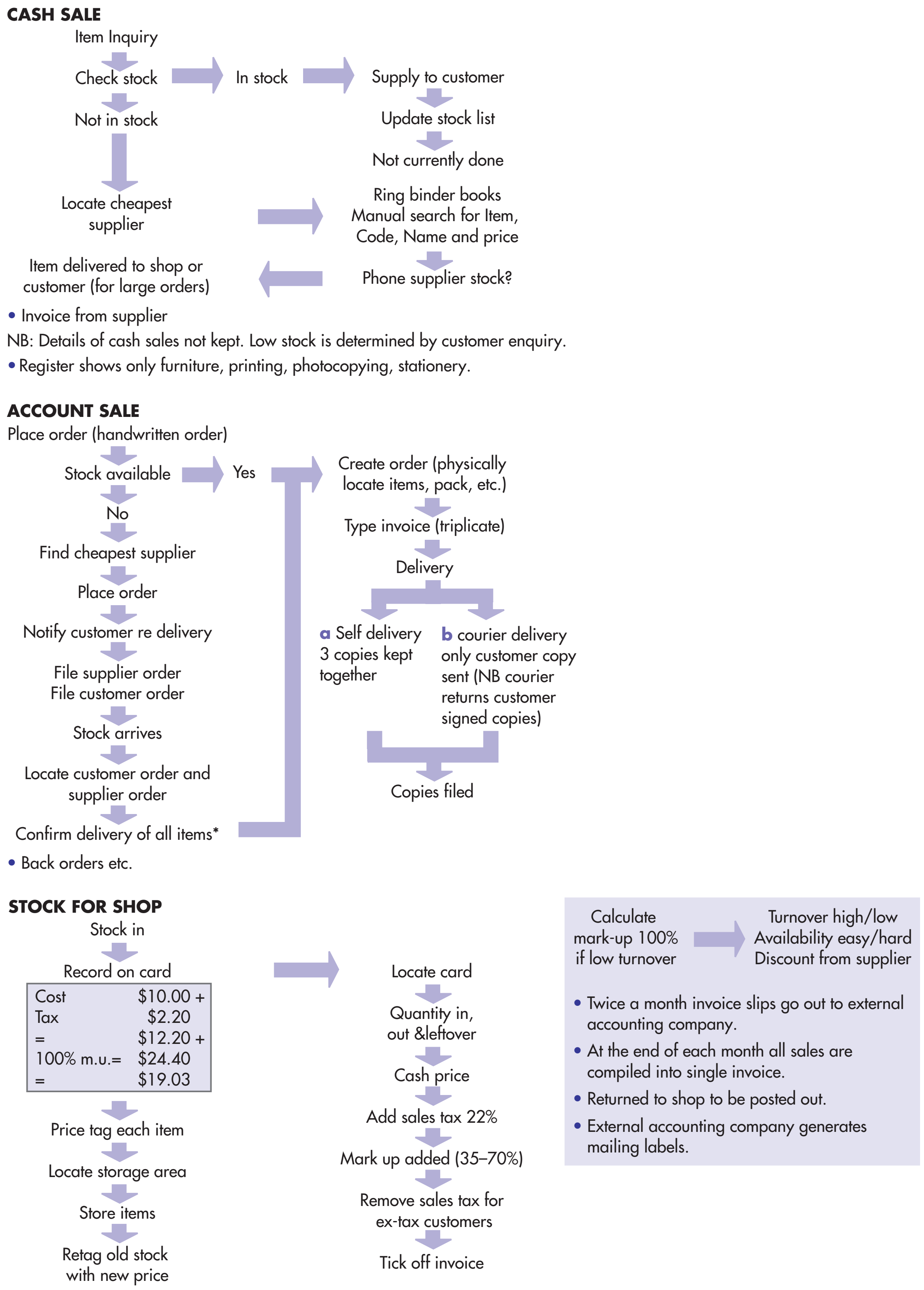
Cash sale process

1. *Customer requests item*
2. *Check stock*
3. *If item is at hand sell to customer*
4. *If item is not in stock:*
5. *Locate cheapest supplier by checking catalogues and files*
6. *Phone supplier and place order*
7. *Item delivered to shop (or direct to customer for large orders)*
8. *Sell item to customer*
9. *Process invoice from supplier*
10. *Pay supplier*

Note: The cash register records only whether an item is furniture, printing, photocopying or stationery. Details of items sold for stock tracking must be done manually.

These handwritten notes need to be converted into dataflow diagrams, IPO charts and algorithms.

Through further communication with the client, we discover that he wants to use Windows NT as his network server. He has had some experience with it in the past and would like to stay with a system that he is familiar with. In addition, he indicates that he wants to use Access as the database software, for similar reasons. It is convenient that the product catalogue is available in Access.


CASH SALE
Item Inquiry
Check stock
In stock
Supply to customer
Update stock list
Not currently done
Not in stock
Locate cheapest supplier
Ring binder books Manual search for Item, Code, Name and price
Phone supplier stock?
Item delivered to shop or customer (for large orders)
ACCOUNT SALE
Place order (handwritten order)
Stock available
Yes
Create order (physically locate items, pack, etc.)
Type invoice (triplicate)
Delivery
a Self delivery 3 copies kept together
b courier delivery only customer copy sent (NB courier returns customer signed copies)
Copies filed
No
Find cheapest supplier
Place order
Notify customer re delivery
File supplier order File customer order
Stock arrives
Locate customer order and supplier order
Confirm delivery of all items*
STOCK FOR SHOP
Stock in
Record on card
Price tag each item
Locate storage area
Store items
Retag old stock with new price
Locate card
Quantity in, out &leftover
Cash price
Add sales tax 22%
Mark up added (35–70%)
Remove sales tax for ex-tax customers
Tick off invoice
Calculate mark-up 100% if low turnover
Turnover high/low Availability easy/hard Discount from supplier

- Invoice from supplier

NB: Details of cash sales not kept. Low stock is determined by customer enquiry.

- Register shows only furniture, printing, photocopying, stationery.

- Back orders etc.

Cost	$10.00 +
Tax	$2.20
=	$12.20 +
100% m.u.=	$24.40
=	$19.03

- Twice a month invoice slips go out to external accounting company.
- At the end of each month all sales are compiled into single invoice.
- Returned to shop to be posted out.
- External accounting company generates mailing labels.

Figure 8.3 Dataflow diagram made from handwritten notes.

Exercise 8.2

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

The three stages of defining the problem are __________, __________ and __________. These are broken down into sub-stages including __________, __________ and __________. The separation into these stages is usually artificial and there is often __________ between the various stages. It is during the problem definition stage that information is gathered about the __________. This involves several __________ with the client and all __________ of the system.

Use the notes in Figure 8.3 to complete the following exercises.

2 From the notes create a data flow diagram for the purchase of items by a customer who has an existing account with the company.

3 Create a data flow diagram that shows how new stock is purchased and added to the warehouse.

4 Create a data flow diagram that describes the trading stock system. You will need to include a facility to alert the owner when certain stock is low and needs replenishing.

5 From the information gathered so far, create a requirements definition for your client.

Understanding the problem

Having defined the problem, it now needs to be examined in greater detail. Preliminary decisions need to be made about aspects of the software such as interface design, diagrammatic representation of the system, selection of appropriate data structures, project management techniques and consideration of all social and ethical issues.

Interface design

At this stage of the development, it is sufficient to consider only whether the user interface is to be primarily text oriented or graphical. This will be influenced by the type of software that is to be used and by the preferences of the client. The client wants to create a web presence. It would be wise to use the web interface for most of the interaction with the system. This is achievable for interaction with the databases, and searching, updating and ordering can all be achieved through a web interface. The accounting package comes with a pre-defined user interface.

Interface design is covered in more detail later in this chapter when the design of the actual webpages is discussed.

Communication with others involved in the proposed system

As stated earlier, communication with others involved in the proposed system is crucial. At this stage it will involve the gathering of detailed information about the current system. This information will be used in the next stage to diagrammatically represent the systems both old and new. In order to gather this information, a variety of techniques is used, including the use of interviews and questionnaires. Where appropriate, a member of the project team may be assigned to spend some time in the business observing the existing system.

Representing the system using diagrams

Having collected data on the existing system and discussed the client's requirements of the new system, it is time to put this information together in a meaningful and useful way. One way to help understand the existing and new systems is to represent them diagrammatically.

Input	Process	Output
Customer details Order details	Order form completed by customer Customer requirements entered Order recorded and passed on to despatch Stock database adjusted	Updated database entry Completed order ready for delivery

Table 8.1 An IPO chart

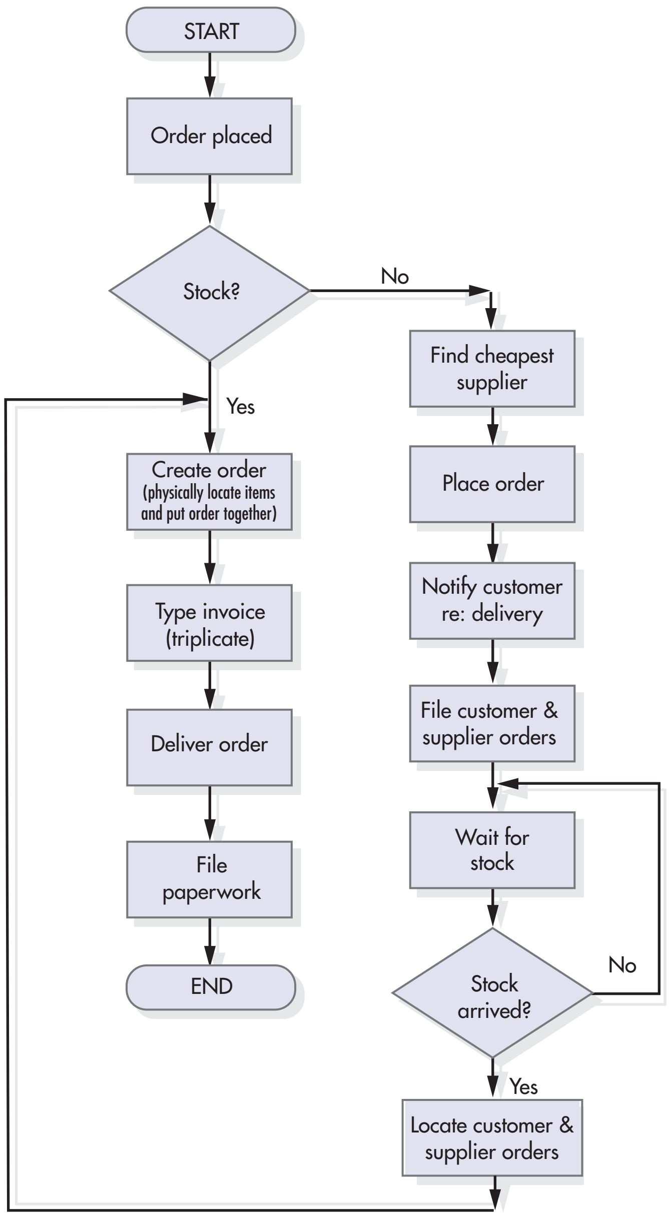

START
Order placed
Stock?
No
Yes
Create order (physically locate items and put order together)
Type invoice (triplicate)
Deliver order
File paperwork
END
Find cheapest supplier
Place order
Notify customer re: delivery
File customer & supplier orders
Wait for stock
Stock arrived?
No
Yes
Locate customer & supplier orders

Figure 8.4 Flowchart describing an account sale.

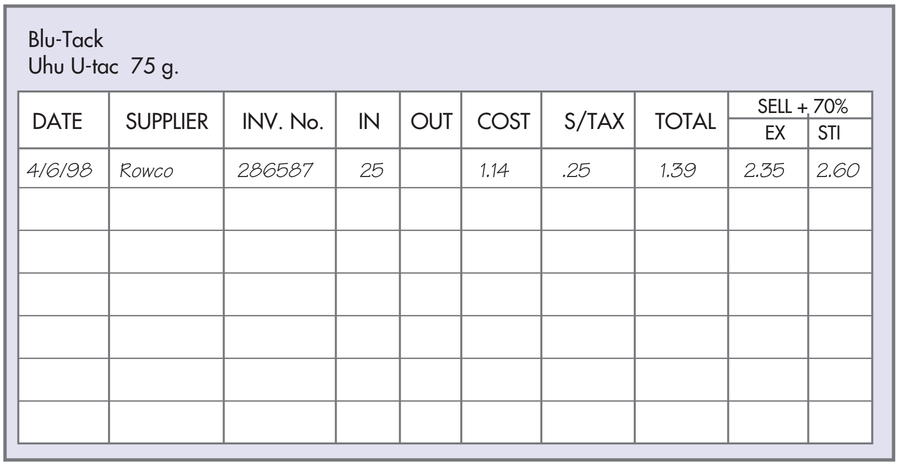
This can be achieved in a number of ways including data flow diagrams, flowcharts, input/output charts and algorithms. An example of an input/ output chart for this solution is shown in Table 8.1.

Algorithms are generated from the handwritten notes. The following simple algorithm is for a cash sale. The underlined parts of the algorithm indicate subroutines that are described in other algorithms.

```
BEGIN Cash sale
Customer enquiry
Check stock
    WHILE item not in stock
    purchase stock
    ENDWHILE
Sell item to customer
Record sale
Update inventory
END Cash sale
```

As well as gathering information on the workings of the current system, the team has collected a variety of forms, catalogues and cards. These will be examined closely as they will influence the decisions regarding the fields required in the database. Figure 8.5 shows a stock record card for the item Blu-Tack. Notice the markup (+ 70%) written next to the 'SELL' heading.

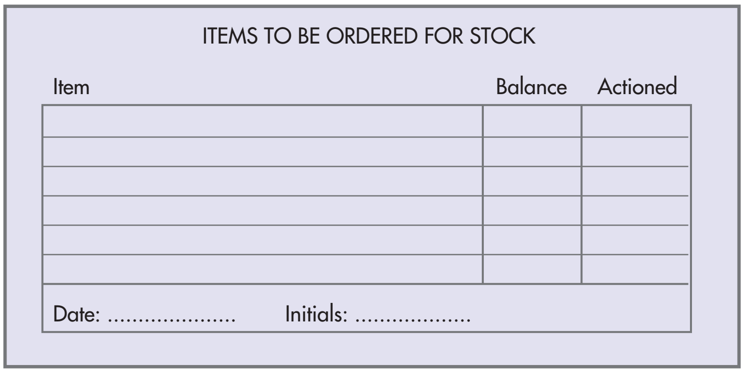
Another of the forms used in the shop is shown in Figure 8.6. This form is used on a daily basis to keep track of items that are sold as stock is becoming low.



Blu-Tack
Uhu U-tac 75 g.

DATE	SUPPLIER	INV. No.	IN	OUT	COST	S/TAX	TOTAL	SELL + 70%	
								EX	STI
4/6/98	Rowco	286587	25		1.14	.25	1.39	2.35	2.60

Figure 8.5 A sample stock record card.



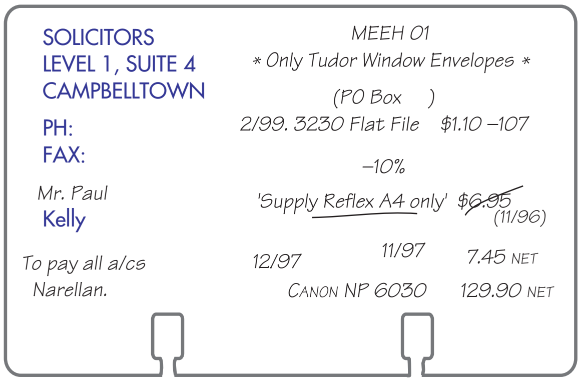
ITEMS TO BE ORDERED FOR STOCK

Item	Balance	Actioned

Date: .... Initials: ....

Figure 8.6 Form used to replenish stock.

Finally, customer details are kept on cards in two rolodexes. These cards contain such information as the customer's name, address and phone and fax numbers and the name of contact people if the customer is a business. You will notice that the customer record card in Figure 8.7 has additional information hand-written on it. These details include the types of things the customer purchases regularly and details of special discounts. This card shows that the customer, a solicitor, uses flat files, uses only Reflex A4 paper, has a Canon NP6030 and usually receives a 10% discount.



SOLICITORS
LEVEL 1, SUITE 4
CAMPBELLTOWN

PH:
FAX:

Mr. Paul
Kelly

To pay all a/cs
Narellan.

MEEH 01
* Only Tudor Window Envelopes *
(PO Box)
2/99. 3230 Flat File $1.10 –107
–10%
'Supply Reflex A4 only' ~~$6.95~~ (11/96)
12/97 11/97 7.45 NET
CANON NP 6030 129.90 NET

Figure 8.7 Example of a customer record card.

Selection of appropriate data structures

When writing the algorithms for our solution, we need to consider the data structures that will be appropriate. Since we are creating two databases for this project, we will need to consider using structured data types rather than simple data structures. Our databases will consist of customer records, as well as records relating to stock. A number of files will be used and they will need to be linked, so a relational database will be used. The product catalogue is in MS Access form, so in order to minimise compatibility problems and to follow the principle of reuse of existing code, we will use MS Access to create the database for this client. The creators of the database management system (DBMS) will have made the bulk of the decisions about data structures. For our purposes, the decisions to be made will be the field types and their interrelationships. These decisions will be informed largely by the nature of the information contained on the examples collected in initial meetings, as well as through consultation with the client.

Consideration of all social and ethical issues

Software is built for people's use. Without users, software becomes a meaningless collection of bits. When embarking on a software development project, consideration needs to be given to the social and ethical issues that relate to both the project development and the finished product.

The relevant issues here include ease of use, accessibility of technical language, copyright and ergonomics.

Ease of use

Ease of use involves the ergonomics and intuitiveness of the software interface. It should be the goal of every user interface designer to make the interface almost disappear, allowing the users to concentrate on their work. Software has to be easy to use. It doesn't matter how fancy the programming is, if the program is difficult to use or doesn't take into account the users' needs it will not fulfil its purpose. When developing software time must be invested in learning the requirements of the software and designing a user interface that is understandable by the potential user. A great deal of effort is required to create an environment in which tasks are carried out effortlessly and users are 'in the flow'.

Accessibility of technical language

When designing the user interface, the use of technical language should be avoided as much as possible. Jargon or language that is too technical is off-putting to the user and can get in the way of what the software is actually trying to achieve. Where technical language is required, it should be accompanied by a brief explanation.

Copyright

Piracy or the illegal use of software is one of the most common computer crimes. Copyright laws cover most software. Every effort must be made to gain the permission of the copyright holder for any material that is used in your software. This includes all graphics, text and modules that are used. It is estimated that software piracy costs the Australian computer industry $400 million per year.

Ergonomics

Since software is the link between the computer and the operator, it should be ergonomically designed to make the operator feel relaxed and comfortable. In designing the webpages and screens for this system, every effort should be made to ensure that the text is easy to read and that navigational elements are in consistent places. Pages and screen transitions should load quickly in order to minimise the amount of time that the user has to wait. For example, the choice of text and background colours can mean the difference between customers persisting and actually placing orders and customers moving away from your website to other more visually pleasant sites. A trend in webpages for some time was to have red text on a black background. This combination is actually quite hard on the eyes and web developers have moved to a more natural white background with black text. Good use of white space also helps to enhance the legibility of the page. This mimics the printed text which is easier to read than text on a computer screen.

Exercise 8.3

1 Copy the following passage and complete it by filling in the blanks with the appropriate terms or phrases.

To understand the problem we need to examine it in greater detail. Steps covered at this stage include __________ design, __________ __________ of the system, selection of appropriate __________ __________, project management and consideration of all __________ and ethical issues. When planning the user interface, __________ of use is critical. Communication with others involved in the __________ system is done regularly and includes as many of the users as possible. Diagrammatic representation of the system includes the generation of __________ __________, algorithms, __________ charts, data __________ diagrams and flowcharts. Decisions need to be made regarding the __________ __________. These structures need to be appropriate to the solution. Consideration of all social and ethical issues consists of examining __________ of use, accessibility of __________ language, __________, __________ and __________ Ergonomics considers the __________ between the computer and the operator.

2 Table 8.1 is a simplified IPO chart for the case study. Create a detailed IPO chart for the solution. You may need to make multiple charts for the various components of the solution.

3 On p. 231 is an algorithm for a cash sale. Write the algorithms for the subroutines as indicated by the underlined elements in the main algorithm.

4 Modify the algorithm given in figure 8.4 to calculate prices using 10% GST instead of sales tax.

5 Using the forms given in figures 8.5, 8.6 and 8.7 as your starting point, develop several possible database entry forms. Do this using pen and paper, not your computer. Discuss your designs with other class members and select the most appropriate one to be coded into your database.

6 Create some preliminary screen layouts for the final solution. These should not be too polished, as they will be modified as you consult with others on their effectiveness in relation to the solution. If you create these on computer, print them out and encourage others to comment on them and mark them up with possible alterations. Remember to consider ease of use in your designs.

Planning and design

Having identified and understood the problem, it is now time to begin the planning and design stage of the development cycle. It is at this stage that ideas are either kept or discarded. Major decisions have been made and nothing new is added to the specifications of the solution. (In other words, no major changes are made. It is impossible to carry through any project without making minor adjustments since it is not possible to anticipate every contingency.)

Interface design

Interface design decisions for the accounting package will be limited to what is available in MYOB. Major decisions regarding interface design will be required for the web interface. The level of interaction with the databases will govern the decisions, in part. The client is unable, at present, to spend the money required to build the software from scratch. Typically, this is in the range of $50 000 and upwards depending on the scope of the development. Some basic guidelines will be adhered to. These include the principle of a minimum number of mouse clicks to find information, mirroring hard copy forms with those used in the software and, as part of that, ensuring that the tabbing order of fields follows naturally from the printed form to the screen. This latter principle means that the operator does not have to look away from the data entry forms to the screens in order to confirm that information is going into the correct fields on the screen. The operator should be able to press 'tab' at the completion of each field entry confident that the next field on screen will match exactly the next screen on the data entry form. (In the new system, the use of data entry forms will be limited since barcode readers will be used to enter stock and record sales.)

Careful thought must also be given to the consistent usage of navigation elements, fonts and styles throughout the database and the webpages linked to it, and the consistent placement of information on the screen.

Selection of software environment

Selection of the software environment depends on a number of factors, including the particular expertise of the programming team and where and how the finished product will be used. Programming environments range from a collection of text editors, linkers and compilers through to a large collection of highly integrated tools, each accessed via a uniform interface. The latter greatly assists in the development and maintenance of software. Unix is an example of a collection of tools with disparate interfaces; Microsoft Visual C++ represents a large and elaborate collection of software development tools with a seamless interface.

As stated earlier, the client wishes to use Windows NT for his networking and MS Access for the databases. The programming environment will vary from client to client depending on a number of factors including customer and system requirements, the availability of existing software that can be adapted and the particular areas of expertise held by the programmers and systems developers. The software development environment for our team will include MS Access, text editors, graphics software, programming software (to create CGI (common gateway interface) scripts etc.) and WYSIWYG HTML editors.

Identification of appropriate hardware

Identification of appropriate hardware needs to be carried out at a number of levels. For example, the client's requirements will mean the purchase of a network server, a web server, modems, hubs, a router, four to five workstations, a barcode scanner, a scanner, a colour printer able to print up to poster size, and interfaces for the EFTPOS and electronic credit card systems.

The team responsible for the creation of the network will identify the bulk of these requirements. Our team, in charge of the databases and webpages, will need to communicate the minimum hardware requirements to the networking team in order for the database services to run smoothly. These decisions will also be constrained by the budget set aside for the purchase of the hardware. It is pointless to plan a $200 000 network if only $30 000 has been set aside.

A diagram of the office and warehouse floor indicating the placement of hardware items will assist in the decision-making process. It will help to locate where items need to be located physically and to determine, for example, such things as the number of barcode readers required. Figure 8.8 shows a possible layout of the shop and warehouse. It is not to scale and is indicative only.

We would need at least two barcode readers, one at the sales counter and one at the workstation in the warehouse area. The latter would be used primarily for recording incoming stock, while the other would be used to process sales and to keep the stock level up-to-date.

Selection of appropriate data structures

In writing the algorithms for our solution, we need to consider the data structures that will be appropriate. Since we are creating databases for this project, we will need to consider using structured data types rather than simple data structures. Our databases will consist of

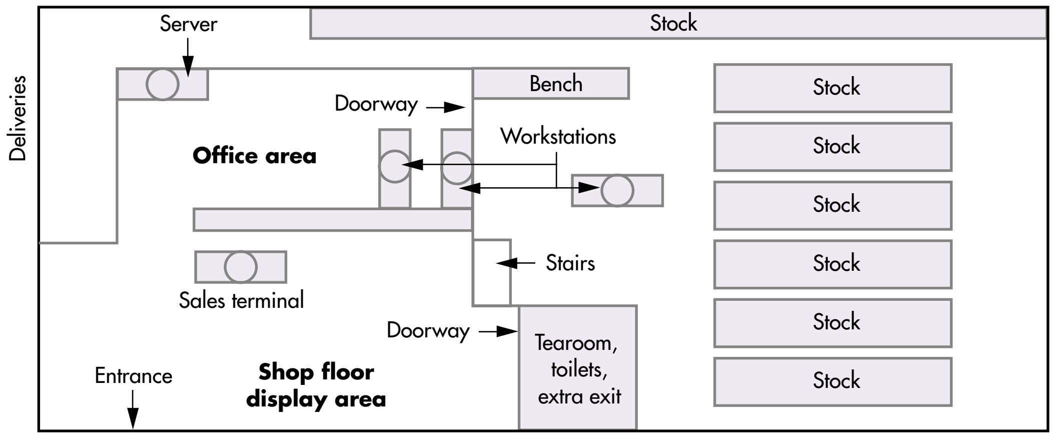


Figure 8.8 Shop and warehouse layout.

customer records and records relating to stock. We will need to use a number of files that will need to be linked, so a relational database will be used. As the client wishes to minimise compatibility issues with the catalogue database supplied by the wholesaler, we will use MS Access to create the database. The creators of the DBMS will have made the bulk of the decisions about data structures. For our purposes, the decisions to be made will be the field types as we create the database files.

Production of a data dictionary

Data dictionaries describing the data used in the program need to be created. Table 8.2 is a possible data dictionary for customer details. We would also need to create a data dictionary for the stock database. This would show fields such as item number, item description, cost price, sales tax, mark-up, quantity in stock and so on.

Field	Type	Size	Range	Example
Surname	String	Up to 20 characters	A to zzzzzzz …	Smith
First_name	String	Up to 15 characters	A to zzzzzzz …	John
Street_address	String	Up to 35 characters	0 to zzzzzzz …	1 First Avenue
Suburb	String	Up to 20 characters	A to zzzzzzz …	Fairfield
Postcode	String	4 numeric characters	0000 to 9999	2165
Phone_no.	String	10 numeric characters	0000000000 to 9999999999	0296574556

Table 8.2 Data dictionary—customer details

Exercise 8.4

1 The data dictionary in Table 8.2 is quite generic. What fields would need to be added in order to more closely represent the type of customer details kept by your client? Hint: look at the customer record cards in Figure 8.6.

2 Create the data dictionaries required for the two databases.

3 From the specified hardware requirements, calculate an approximate costing for the hardware. Use current PC magazines for approximate prices.

4 Using Figure 8.8 as your starting point, draw a plan of the warehouse showing where the computer terminals and servers will go. Also show items such as telecommunications

lines. Take into account such things as workflow and display areas for the shopfront part of the business. Remember that the owner also wishes to include a training area in the future. He intends to use the area above the showroom for this.

Definition of required validation processes

In order for our solution to be robust and operate efficiently, we need to include validation processes. This can include checking that data is entered into cells where there must be data and that data is of the correct type, meaning that it falls within a predetermined range of acceptable data. For our solution, the developers of the DBMS and of the accounting package have done much of this for us. We still need to ensure that data entered into the forms on our webpages is valid.

We need to validate user input. For example, when placing an order we need to be able to validate that each field has input entered that falls within a certain range. Data entered into the telephone number field can only be eight digits long; characters other than numbers are not allowed.

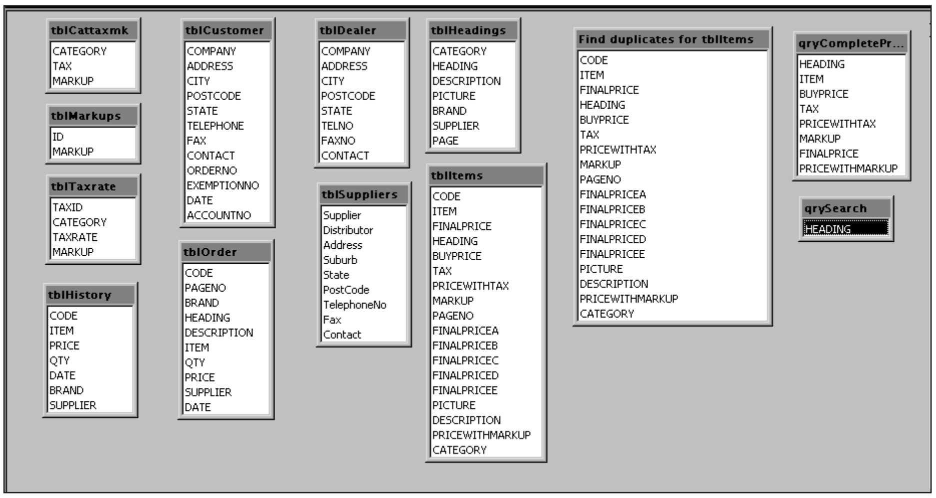
The server can validate our forms or the user's computer can carry it out. The former is called server-side validation, while the latter is called client-side validation. In server-side validation the form is submitted to the server, checked for errors and then returned to the user's computer to be resubmitted if errors are found. Client-side validation processes the form on the user's machine, prompting the user to correct any errors before the form is submitted to the server. This second method is more desirable, as it can give the user immediate feedback and reduces network traffic. We will use JavaScript to achieve client-side validation.

Definition of files—record layout and creation

Definition of the files involves looking at the type of data to be stored. We need to determine what files (often called tables in some DBMS packages) will be needed in the database. These files will hold unique data and will be linked to each other by the use of common fields. The database provided by the wholesaler has already done this. Figure 8.8 shows the tables in the product database.

Algorithm design

We need to design the algorithms that will later be converted to code. A number of algorithms will be required. We will need algorithms for the different types of sales that are


tblCattaxmk
CATEGORY
TAX
MARKUP

tblMarkups
ID
MARKUP

tblTaxrate
TAXID
CATEGORY
TAXRATE
MARKUP

tblHistory
CODE
ITEM
PRICE
QTY
DATE
BRAND
SUPPLIER

tblCustomer
COMPANY
ADDRESS
CITY
POSTCODE
STATE
TELEPHONE
FAX
CONTACT
ORDERNO
EXEMPTIONNO
DATE
ACCOUNTNO

tblOrder
CODE
PAGENO
BRAND
HEADING
DESCRIPTION
ITEM
QTY
PRICE
SUPPLIER
DATE

tblDealer
COMPANY
ADDRESS
CITY
POSTCODE
STATE
TELNO
FAXNO
CONTACT

tblSuppliers
Supplier
Distributor
Address
Suburb
State
PostCode
TelephoneNo
Fax
Contact

tblHeadings
CATEGORY
HEADING
DESCRIPTION
PICTURE
BRAND
SUPPLIER
PAGE

tblItems
CODE
ITEM
FINALPRICE
HEADING
BUYPRICE
TAX
PRICEWITHTAX
MARKUP
PAGENO
FINALPRICEA
FINALPRICEB
FINALPRICEC
FINALPRICED
FINALPRICEE
PICTURE
DESCRIPTION
PRICEWITHMARKUP
CATEGORY

Find duplicates for tblItems
CODE
ITEM
FINALPRICE
HEADING
BUYPRICE
TAX
PRICEWITHTAX
MARKUP
PAGENO
FINALPRICEA
FINALPRICEB
FINALPRICEC
FINALPRICED
FINALPRICEE
PICTURE
DESCRIPTION
PRICEWITHMARKUP
CATEGORY

qryCompletePr...
HEADING
ITEM
BUYPRICE
TAX
PRICEWITHTAX
MARKUP
FINALPRICE
PRICEWITHMARKUP

qrySearch
HEADING

Figure 8.9 Tables in the product database.

processed, for example cash sales, account sales and sales involving back orders. We will also need algorithms for updating the stock database and for adding new customers to the customer database. These are relatively high-level algorithms and will consist of smaller algorithms, such as calculating the total cost of an order (which will depend on the type of sale and so on). For example, how do we calculate the cost of an order for a tax-exempt customer who usually receives a 10% discount on paper products and a 20% discount on other stationery? The following short algorithm determines the taxable status of a customer.

```
BEGIN Taxable_Customer
READ Customer ID
READ Field Taxable
    WHILE Field Taxable = true
    ADD 22% sales tax
    ENDWHILE
    Do not add 22% sales tax
END Taxable_Customer
```

Inclusion of standard or common routines

One advantage of the structured approach to software development is that problems are broken down into easily manageable parts. The parts are solved individually and modules are created that can be reused in future programs. The developers need to identify any existing code that may be used, thereby saving the customer time and money. This methodology also lends itself to the use of standard or common routines. For example, an algorithm for calculating the cost of an order to a customer uses common routines that have been developed and refined elsewhere. These can be adapted for this solution. As shown in Figure 8.9, if we are to reuse code then the appropriate modules must be used.

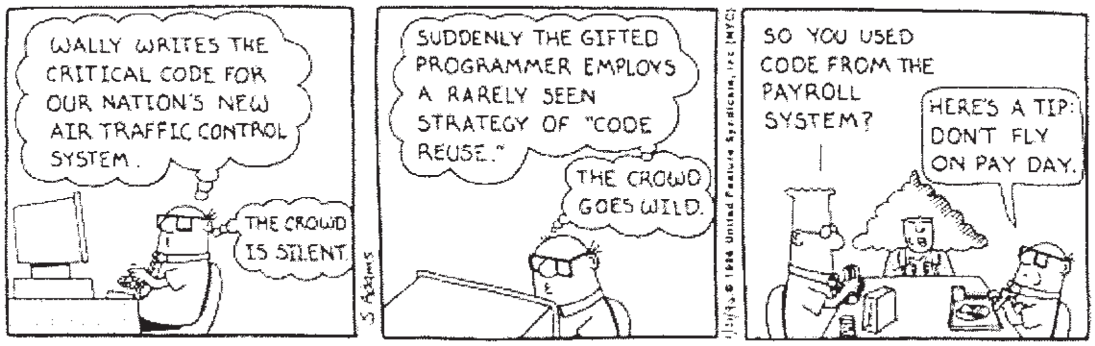

WALLY WRITES THE CRITICAL CODE FOR OUR NATION'S NEW AIR TRAFFIC CONTROL SYSTEM.
THE CROWD IS SILENT.
SUDDENLY THE GIFTED PROGRAMMER EMPLOYS A RARELY SEEN STRATEGY OF "CODE REUSE."
THE CROWD GOES WILD.
SO YOU USED CODE FROM THE PAYROLL SYSTEM?
HERE'S A TIP: DON'T FLY ON PAY DAY.

Figure 8.9 Appropriate modules must be selected when reusing code.

Use of software to document design

A vast amount of documentation will be generated throughout the system development cycle. The documentation will generally be of two types: product documentation and process documentation. The process documentation will mostly become outdated. The product documentation will take one of two forms: system documentation and user documentation. There are various software packages available that allow us to create this documentation relatively easily. The simplest of these is the integrated packages of word processor, spreadsheet, database and drawing/painting programs. These can be used to generate most types of documentation. There are, however, many purpose-built packages available for various aspects of documenting the design process. Microsoft Project is used to track the design process, flowcharting software can be used to create flowcharts, and presentation software can be used to prepare briefing sessions for members of the project team and other stakeholders such as senior management and the client.

Identification of appropriate test data

In order to create algorithms that actually do what they are designed to do, we need to identify appropriate test data that can be processed by the algorithms. This stage is referred to as desk checking, as it is performed using pencil and paper before any code is generated. We need to identify appropriate test data for each of our algorithms. The test data should test all parts of the program, testing each of the paths that can be taken during execution and each side of any boundary value as well as the value itself. For example, our client's customers can receive a variety of discount values. Some receive 10%, some 15% and some 20%. There are also customers who receive no discount. Our test data would have to test these values and their boundaries. We would need to ensure that negative values, –5%, and high values, >20%, are not accepted as valid inputs. Table 8.3 shows some sample test data.

Discount	Expected output
0%	Acceptable
9%	Invalid input—not 0%, 10%, 15% or 20%
10%	Acceptable
11%	Invalid input—not 0%, 10%, 15% or 20%
14%	Invalid input—not 0%, 10%, 15% or 20%
15%	Acceptable
16%	Invalid input—not 0%, 10%, 15% or 20%
19%	Invalid input—not 0%, 10%, 15% or 20%
20%	Acceptable
21%	Invalid input—not 0%, 10%, 15% or 20%
100%	Invalid input—not 0%, 10%, 15% or 20%
–50%	Invalid input—not 0%, 10%, 15% or 20%

Table 8.3 Sample test data for discounts

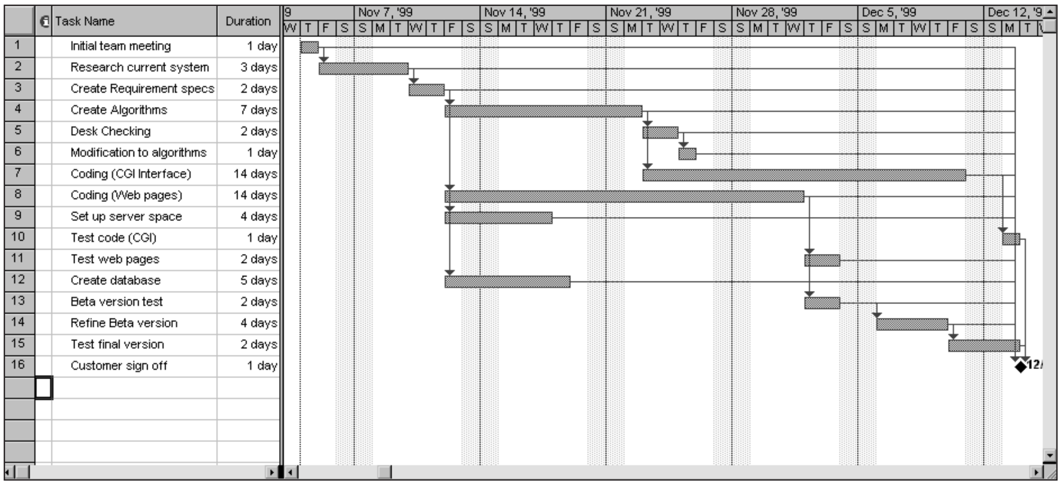
Enabling and incorporating feedback from users at regular intervals

As algorithms are developed and modules are completed, they should be presented to all users of the new system in order to gain feedback and to allow appropriate modifications to be made. This is a most important step and it is crucial that it is done at regular intervals, as it is easier and cheaper to rewrite a module rather than an entire software package.

Applying project management techniques

Applying project management techniques involves creating a project team and assigning tasks. Other things that need to be considered are time-lines and resources. We need to consider the resources we have available to us. This includes physical resources such as hardware and software as well as non-physical resources such as time. It is the task of the project manager to ensure that all tasks are completed within specified deadlines, using only the resources allocated. The project manager will have a variety of project management resources available. These include Gantt charts, PERT charts and so on. Many of these resources are available in software packages such as Microsoft Project. Figure 8.10 shows a Gantt chart created using MS Project.

It is essential that the project manager ensure that all team members keep logs and document everything they do. This is most important, as over the life of the project team members will leave, for promotions or career changes, and new team members will come in. The use of logs and documentation will enable the new team members to quickly familiarise themselves with the project.



	Task Name	Duration
1	Initial team meeting	1 day
2	Research current system	3 days
3	Create Requirement specs	2 days
4	Create Algorithms	7 days
5	Desk Checking	2 days
6	Modification to algorithms	1 day
7	Coding (CGI Interface)	14 days
8	Coding (Web pages)	14 days
9	Set up server space	4 days
10	Test code (CGI)	1 day
11	Test web pages	2 days
12	Create database	5 days
13	Beta version test	2 days
14	Refine Beta version	4 days
15	Test final version	2 days
16	Customer sign off	1 day

Figure 8.10 Gantt chart.

Exercise 8.5

1. Create the Gantt chart for this case study project. If you have access to project software, use it; otherwise use pen and graphing paper. Refer back to the Gantt chart that you created for question 4 in exercise 8.1. The Gantt chart for this exercise should be quite detailed. You will need to show all milestones, as well as resources required and when they will be available.
2. Create all of the algorithms required for this project. You may represent these as pseudocode or as flowcharts.
3. Create sets of test data to thoroughly test all the algorithms.
4. Discuss the use of a variety of software packages to document the development process.
5. Select a software package that will help you to document the development process. Justify your choice.

Systems implementation

It is at this stage that the actual system is created. This involves the conversion of algorithms into code, the documentation of the system and the conversion from the existing system to the new one. The process consists of two main tasks: implementation and maintenance. These are in turn broken down into sub-tasks. This section looks at each of these as they relate to our case study.

Implementation

Having completed the analysis and design stages of the software development cycle, we are now ready to enter the implementation stage. In this stage we turn algorithms into code. We actually create the modules using our chosen programming language. We then test each module, create the relevant documentation and, finally, put the system together and get it running. In this case study the modules that our team is creating are the two databases and the webpages. Once the system is in place and performing as per the specifications, the only thing remaining is to perform regular maintenance on the system. This includes testing and modifying the software as required.

Production and maintenance of data dictionary

The data dictionaries are created at this stage and they need to be kept up-to-date with any changes to the type of data that will be used in the program.

Use of software to document design

As mentioned earlier, there are many software packages available to assist in the documentation of the design process. Most of the tools that are available to software developers or project managers belong to one of five categories.

- Project management tools such as Microsoft Project help a team to estimate, plan and track schedules, resources, effort and costs.
- Analysis and design tools assist in the documentation, analysis and management of requirements or in the creation of design models.
- Coding tools include code generators and reformatters and code analysis and reverse engineering tools.
- Quality improvement tools include test planning and execution tools and static and run-time code analysers.
- Configuration management tools allow you to track changes and defects, to control access to files and to build a product from its components.

These tools help to save time and reduce errors by automating a part of the development and management process. An important thing to remember is that these tools only help to implement processes; they are not a replacement for having established processes.

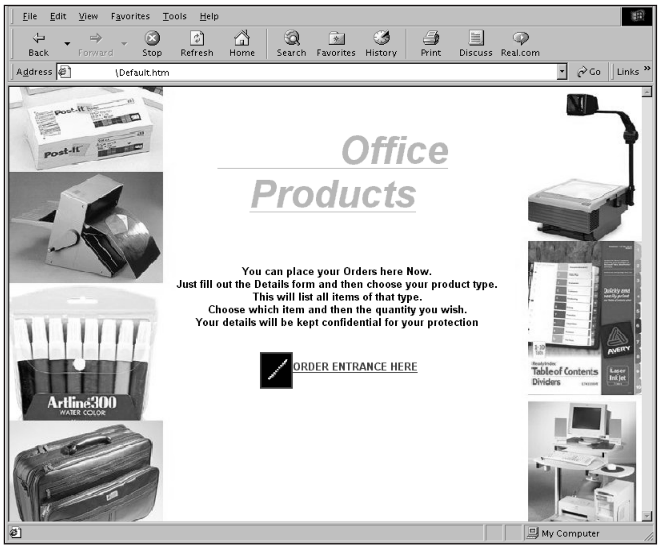
Other specialised software tools that help with this task and with the task of writing code are called CASE tools. The acronym 'CASE' stands for 'computer-aided software engineering'. CASE tools are used predominantly for systems analysis and design. For a more thorough explanation of CASE tools, see Chapter 2.

Translating the solution into code

Once we have designed the solution fully and all desk checking is complete, it is time to begin generating code for the implementation of the solution. The coding of the product catalogue has been completed. It is contained in the database supplied by the wholesaler. The database contains product names and numbers and needs to have added to it other information such as price, mark-up, GST and so on. The client database needs to be created and work needs to begin on creating the webpages.

For the webpages, we need a static entry page. Figure 8.11 shows a preliminary design for the entry page. This page contains general information such as the company name and address. It will have links to other pages such as help pages, search pages and ordering information pages.

As most of the customer interaction will involve searching for specific items, most of the pages will be created automatically by our database. Templates only will be required for these pages as content will be drawn from the database and inserted in the appropriate sections of the templates.



You can place your Orders here Now.
Just fill out the Details form and then choose your product type.
This will list all items of that type.
Choose which item and then the quantity you wish.
Your details will be kept confidential for your protection

Figure 8.11 Preliminary design for entry page.

Creating online help

In order to make the customers' experience as pleasant as possible, we will add some online help to our web pages. This will take a number of forms. One of these will be to have a help section for our website. The customer can click on 'help' and will be taken to a series of pages that explain in simple terms how the site works, what processes are required to place an order, and with time a collection of frequently asked questions with their answers. Another way of incorporating online help in webpages is to create JavaScript rollovers. When the user's mouse rolls over a certain element on the screen a small help message will appear telling the user what that element does. For example, the navigation buttons will display a short message telling users what they will be taken to if they click on that button.

Program testing

Program testing is one of the most important steps in any software development. If you spend thousands or hundreds of thousands of dollars developing a piece of software, you want to be sure that it works according to the original requirements specification. In order to minimise the amount of recoding required, the software package is broken down into smaller modules. As each module is completed, it is tested. Stubs, small pieces of code that take the place of modules yet to be coded, are used to test each completed module. Once we have created the databases and the webpages, as many people as possible from as many different disciplines as possible need to test them.

Reporting on the status of the system at regular intervals

An aspect of good project management is to ensure that there is reporting on the status of the system at regular intervals. This should not be allowed to happen in an ad-hoc fashion. Regular meeting times should be scheduled at the outset of the project. These meetings should be organised so as to coincide with imminent milestones in the project. That is, the meetings should occur at or before a critical time such as before the purchase of major equipment. There should be a meeting to ensure that the correct hardware has been specified. It would be pointless to order the hardware and then hold a meeting to determine whether the appropriate hardware has been ordered. These regular reporting sessions are designed to ensure that there are no 'surprises'. The time-line of this project is eight weeks. Over this period of time, there should be a report at the very least at the end of each week. This will ensure that the project remains on track and that any potential problems can be anticipated and acted upon.

Completing all user documentation for the project

Having been coded and tested, the modules need to be combined into the final system. An important part of this process is the completion of all user documentation for the project.

User documentation can include user manuals, installer manuals and other types of help manuals and documentation that relate to the software. User manuals are usually written after the software is complete and the focus is to show the end user how to use the program. Installer manuals are written chiefly for systems administrators and will guide these people through the installation process. This may include instructions on how to load the software onto stand-alone machines, network servers and networked workstations. Other types of manuals may include the internal documentation in the software itself. This is for use by programmers who may need to debug or rewrite the software, and is usually in the form of comments throughout the code of the program. The computer ignores these comments as they are intended to be read by humans. Special characters that tell the computer to ignore them usually precede the comments. For example, in Java any such comments are preceded by the hash symbol (#). In the following two lines of Java code, the first line is the comment and the second is the code that is executed.

```
# Get the date
chomp ($date = '$DATE');
```

Systems documentation includes documents such as algorithms and data flow diagrams that show how the overall system works. The purpose of systems documentation is to provide a detailed description of the system and to provide information that will assist with the maintenance of the system.

Completing full program and systems testing

On completion of all of the system components, they need to be brought together. Each module of the system has been tested during its development, and it is now time to bring all the modules together and test the system in its entirety.

Once the system has been tested, it is time to put it into place. There are a number of conversion options—converting from the existing system to the new one—available to us. These are phased conversion, direct conversion, pilot conversion and parallel conversion.

In phased conversion both systems are run simultaneously. All operations are carried out in both systems. This allows further testing of the new system using a full set of data under realistic conditions. A disadvantage of this type of conversion is that the workload is doubled as all processes are carried out in both systems.

In direct cut-over the old system is taken off line and the new system is put in its place. All data needs to be converted for use with the new system. Staff need to be trained in the use of the new system. A disadvantage of direct conversion is that if the new system fails the old system is not available as a backup.

Pilot conversion involves the entire system being installed but only part of the new system being used. For example, the customer database may be brought online while other tasks are conducted using the existing system. Only a small amount of data is lost if the new system fails.

Phased conversion involves the gradual implementation of the new system. Parts of the new system gradually replace the existing system until the new system takes over completely. If problems arise, only that module of the system is affected.

In this case study phased conversion will be used. Since the existing card system is quite extensive and available staff for the switch over is limited, it was decided to bring the new system online gradually. In the initial implementation of the new system the electronic catalogue will be used to assist in tracking stock levels. At the same time the customer records kept on cards will be entered into the customer database. New customer details will be recorded directly into the database and no new cards will be created.

Maintenance

Once the solution is delivered and put into place, it will need to be constantly monitored and evaluated. Close comparisons will be made with the computerised system and the manual system. It is not economically viable, for example, to provide the web service part of the solution if only a handful of people use it. The owners are hoping that they will gain a significant amount of business through the web in order for the expense to be justified.

Further evaluation will also tell the owners which areas of the system are working well and which are not. There will need to be some adjustment to the system in order to maximise profits.

Modifying the project to ensure an improved solution

Once the solution is in place, it will be monitored closely. The owners will be keen to see if there is any increase in their business from the new website. In addition to this, as the owners monitor the 'final' solution they may well decide that certain aspects do not look or work as they had anticipated and there may be a need to make slight modifications. Some of these modifications may include changing the database over to a purpose-built database, that is, one that is programmed 'from the ground up' specifically for this client.

Review exercises

1 Create the database(s) required for this solution. Your database should contain at least 5 customers and at least 20 items of stock.

2 Create the webpages for the Internet part of the solution. You can use Figure 8.12 as a starting point. The webpages for this exercise can be static catalogue pages that show the types of stock kept by your client. Extension: Link these webpages to your database and incorporate an online ordering system that keeps track of customer selections. You can get ideas on how this type of site works by going to any of the various e-commerce sites on the web. For example, try http://www.dymocks.com.au or http://www.amazon.com.

3 Create HTML help pages for your client. These pages should help a variety of users of the system including employees and customers. Create at least five webpages.

4 Create a set of instructions that show the owner of the new system how to add customers to the customer database.

5 Create a webpage for this solution. In your source code include documentation that tells other programmers what the page does and how it works. For example, if you choose to use JavaScript you should include appropriate comments throughout your code.

6 Create a proposal that explains how the new system will be maintained and improved. You should include recommendations for future enhancements to the system. Justify your recommendations.

Chapter summary

- Project management techniques are applied to the development of a solution package.
- Project teams are multi-disciplinary and outsourcing is employed as required.
- Defining the problem consists of identifying the problem, ideas generation and communication with others involved in the proposed system.
- Understanding the problem consists of interface design, communication with others involved in the proposed system, representing the system using diagrams, selection of appropriate data structures and consideration of all social and ethical issues.
- Consideration of all social and ethical issues includes ease of use, accessibility of technical language, copyright and ergonomics.
- Planning and design consists of interface design, selection of software environment, identification of appropriate hardware, selection of appropriate data structures, production of a data dictionary, definition of required validation processes, definition of files, algorithm design, inclusion of standard or common routines, use of software to document design, identification of appropriate test data, enabling and incorporating feedback from users and applying project management techniques.
- The final stage in developing a solution package is to implement the solution. This is a two-stage process consisting of implementation and maintenance.
- Implementation consists of production and maintenance of the data dictionary, translating the solution into code, creating online help, program testing and completing all user documentation for the project.
- Maintenance consists of modifying the project to ensure an improved solution.
- Project management techniques are applied throughout the development of a solution package.
- Testing of the solution should be carried out at regular intervals—test early and test often.

chapter 9

Evolution of programming languages

Outcomes

A student:

- differentiates between various methods used to construct software solutions (H 1.2)
- describes the historical developments of different language types (H 2.1)
- explains the relationship between emerging technologies and software development (H 2.2)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- identifies needs to which software solutions are appropriate (H 4.1)
- applies appropriate development methods to solve software problems (H 4.2)

Students learn about:

Historical reasons for the development of the different paradigms

- a need for greater productivity
- recognition of repetitive standard programming tasks
- a desire to solve different types of problems (e.g. AI)
- the recognition of a range of different basic building blocks
- emerging technologies

Basic building blocks

- variables and control structures (imperative)
- functions (functional)
- facts and rules (logic)
- objects, with data and methods or operations (object-oriented)

Effect on programmers' productivity

- speed of code generation
- approach to testing
- effect on maintenance
- efficiency of solution once coded
- learning curve (training required)

Paradigm specific concepts

- logic paradigm
  - Prolog, expert system shells
  - heuristics
  - goal
  - inference engine
  - backward/forward chaining
- object-oriented programming
  - C++, Delphi, Java
  - methods
  - classes
  - inheritance
  - polymorphism
  - encapsulation
  - abstraction
- functional (e.g. LISP, APL)
  - functions

Students learn to:

- recognise representative fragments of code written in a particular paradigm
- differentiate between the different paradigms
- evaluate the effectiveness of each paradigm in meeting its perceived need
- identify an appropriate paradigm relevant for a given situation
- interpret a fragment of code, and identify and correct logic errors
- modify fragments of code written using an example of a particular paradigm to reflect changed requirements
- for current and emerging languages, identify an appropriate paradigm

Introduction

A topic that confuses budding programmers is why there are so many programming languages and why, when computers have been programmed for so long, software has to be constantly rewritten, updated and patched. Why is it that, as hardware has become more powerful and reliable, software has seemingly become more error-ridden?

When studying software development paradigms we need to look at the origins of programming languages, where and why they were developed, the pressures the developers were under when they wrote these languages, whether the languages were written for a specific purpose and the expectations of these languages.

This chapter looks at these questions and also tracks the evolution of the various paradigms and their associated languages.

Since the advent of computers, the complexity of both the machines and the tasks they are expected to perform has increased. Modern computers and their applications have totally changed from what they were 10 years ago, and will have totally changed again in 10 years hence. How do developers who create the software for this new technology work in the most efficient and productive way possible?

When new technology becomes available, users and developers at first tend to do the same as they have always done, except now they do it more cheaply and faster. It takes a while for new applications to be developed for new technology.

New languages can be launched with so much surrounding hype and claims as to their uses that the general public comes to expect more from their capabilities. Fuelling this expectation is the increasing use of computers in our society. Is there anything they cannot do?

In the past 10 years the decreasing cost of memory, the increasing speed of processors and increasing miniaturisation has led to the development of programs incorporating features such as voice recognition, mixed text and graphics and an improved human interface. Each of these applications requires a specific solution.

With the development of more powerful computers the task of software development should become an error-free and much easier task. The development of tools to take advantage of these processors could lead to a situation different from today where there is a lot of white box testing.

The development of hardware and the associated software has led to the emergence of high-level languages, which are more readable, reusable and portable.

Research is continuing into the best way to cope with the advances in hardware and how to gain the greatest advantage from them. Are the existing paradigms suitable or should we develop others that are more able to take advantage of new technologies?

The evolution of programming languages is littered with answers to these and other questions that have faced programmers using the new technologies.

Each new paradigm, as it has been developed, has attempted to improve the readability, simplicity, reliability, reusability, cost effectiveness and portability of new applications.

A study of the evolution of programming languages shows how these factors have developed and may give us some clues on how languages will evolve in the future.

Generations of programming languages

The development of computer programming languages closely follows the development of computer hardware. As the capabilities of computers increase, so does the need for more sophisticated programming methods.

Computer languages can be categorised in five generations. The first and second generations of language are known as low-level languages and are processor dependent; that is, they are used to develop programs that are specific to a particular type or series of processor. Third-generation and later languages can resemble either natural languages such as English or symbolic languages such as mathematics. Languages of the third generation and later are used to develop programs in terms of the problem being solved rather than the hardware on which the solution is implemented.

First-generation computer languages were, by necessity, those that could be directly ‘understood’ by computers. They were in binary form. Early computers were programmed using paper tape with holes punched to represent 0s and 1s or they had their instructions wired by means of plugboards, wires and switches. Both methods presented problems when a new program was required and a high level of skill was needed to create the programs. Since the instructions are in binary form, these languages are known as machine languages and are specific for the type of computer being programmed.

Second-generation languages, or symbolic assembly languages, replaced the sequences of binary digits with mnemonic codes (or short code words) to represent instructions. Like machine languages, assembly languages are specific for the type of processor. However, they offer great advantages over machine code, as the mnemonics are easier to remember and read. Their development meant that programming accuracy was improved, since the instructions could be coded using normal written characters. Furthermore, a program did not have to be rewritten if the physical location of a variable or instruction needed to be changed, as the memory locations used to store values are addressed by symbolic names instead of locations. The assembler provides suitable physical memory locations when the program is assembled into machine code for execution. Assembly languages were given names such as AUTOCODER or SAP (symbolic automatic programming).

Third-generation languages provided a great leap forward as they allowed programmers to write programs that were independent of the machine being used or the arrangement of registers and the instruction set of the processor. BASIC, COBOL, ALGOL and FORTRAN are the most widely known third-generation languages. These languages are distinguished from later generations as their structure consists of a sequence of steps, branches and loops. Unlike the second-generation language, where one coded step becomes one machine instruction, third-generation instructions are usually compiled to several machine instructions.

Fourth-generation languages are more difficult to separate from their third-generation ancestors as they may contain some of the same structures. However, in addition to these structures they employ other mechanisms such as screen interaction, form filling and computer-aided graphics. Many fourth-generation languages depend on a database and its data dictionary, as well as extensions of the dictionary which contain logic and business rules. (This extended form of the data dictionary is often known as an encyclopedia.) Fourth-generation languages are concerned more with what needs to be done rather than how it is going to be done, often accomplishing this by forming a software application that can be customised by a user with very little technical expertise. The most common of these languages are sophisticated spreadsheet packages such as *Excel* and *Lotus 1-2-3*, and database management systems such as *Access*, *FoxPro*, *Quattro Pro* and *Dbase*.

The fifth-generation languages allow the programmer to code complex knowledge, from which the computer can draw inferences. The languages of this generation use the disciplines of knowledge-based systems, expert systems, inference engines and language processing which have originated in the field of artificial intelligence. Programs written in a fifth-generation language often appear to be highly intelligent or to possess an expertise much greater than that of most people. Japan’s fifth-generation project, ICOT, is an attempt to use artificial intelligence techniques together with new hardware designs, such as massive parallel processing, to rapidly advance the power of computing technology. This project is important in the development of the fifth generation as there is a coordination of efforts in all research areas, so that new hardware development is made available to the software researchers and vice versa.

Characteristics of each language type

First-generation (machine) languages

A processor can immediately decode first-generation languages. As the processor can only work with binary digits, this is how programs in first-generation languages are coded. Effort and errors in writing these programs have been reduced by the use of bases other than binary when coding. The most common bases used are hexadecimal and octal.

Of further assistance to the programmer is the use of mnemonics to represent instructions.

For example, the instruction 'Clear the accumulator, add the contents of location 102 to it' in binary digits may be coded as:

```
1 0000000 001 1 0000 00000001 1 1 0001 1 0
```

or, using hexadecimal values to represent the bytes, as:

```
80 30 01 C6
```

or, using mnemonics to represent the instructions together with hexadecimal notation for the numerical values and memory locations, as:

```
CLRAC 30 01 C6
```

(This code can be interpreted, for later comparison, as setting *TOTAL* to 0, then adding *MARK* to *TOTAL*.)

Many low-level programmers, especially hobbyists who write small programs, may write their programs in assembly language and enter them manually in hexadecimal. An advantage of a program written in this form is that it can be run without the need for an assembler. Hand assembly can be especially tedious for processors with a very large instruction set because of the number of instructions required to be searched before the necessary coding is found, as well as the physical size of the coded instruction which may extend to six or eight bytes. Machine code is also more difficult to use on computers that

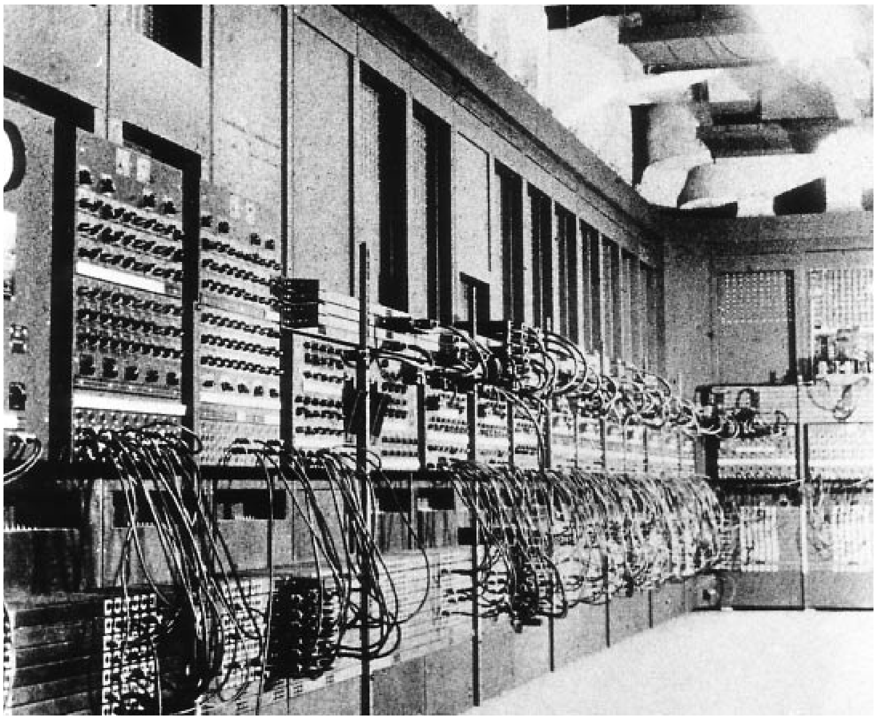


Figure 9.1 Early computers had their instructions wired by means of plug boards, wires and switches.

have a big word size and/or large primary storage, since these words and addresses may take up to 32 bits to describe.

Table 9.1 gives a summary of the assembler mnemonics of the Motorola 68000 microprocessor instruction set. The instructions presented in the table can be combined with any one of 14 different addressing modes and five different data types to effectively create a set of over 1000 different multi-byte instructions.

Each machine instruction can be put into one of five categories covering data transfers, data processing, testing and branching, input and output, and control.

Data transfer instructions are employed to move data between registers and primary storage, and between a register and an input or output device. Some specialised instructions are provided to serve a specific purpose, such as push and pop instructions, which will operate on the stack.

Mnemonic	Description	Mnemonic	Description
ABCD	Add decimal with extend	MOVEM	Move multiple registers
ADD	Add	MOVEP	Move peripheral data
AND	Logical AND	MULS	Signed multiply
ASL	Arithmetic shift left	MULU	Unsigned multiply
ASR	Arithmetic shift right	NBCD	Negate decimal with extend
Bcc	Branch conditionally	NEG	Negate
BCHG	Bit test and change	NOP	No operation
BCLR	Bit test and clear	NOT	Ones complement
BRA	Branch always	OR	Logical OR
BSET	Bit test and set	PEA	Push effective address
BSR	Branch to subroutine	RESET	Reset external devices
BTST	Bit test	ROL	Rotate left without extend
CHK	Check register against bounds	ROR	Rotate right without extend
CLR	Clear operand	ROXL	Rotate left with extend
CMP	Compare	ROXR	Rotate right with extend
DBcc	Test condition, decrement and branch	RTE	Return from exception
DIVS	Signed divide	RTR	Return and restore
DIVU	Unsigned divide	RTS	Return from subroutine
EOR	Exclusive OR	SBCD	Subtract decimal with extend
EXG	Exchange registers	Scc	Set conditional
EXT	Sign extend	STOP	Stop
JMP	Jump	SUB	Subtract
JSR	Jump to subroutine	SWAP	Swap data register halves
LEA	Load effective address	TAS	Test and set operand
LINK	Link stack	TRAP	Trap
LSL	Logical shift left	TRAPV	Trap on overflow
LSR	Logical shift right	TST	Test
MOVE	Move	UNLK	Unlink

Table 9.1 Motorola 68000 instruction set summary.

Data processing operations fall into the five categories of arithmetic operations, bit manipulation (such as setting a flag), increment/decrement (adding or subtracting 1 from a register), logical operations (such as AND, OR) and shift/skew operations (such as a shift left or right).

Test instructions are provided to test various bits in one or more registers (for example the flag register which contains a number of bits each indicating an effect such as the result of a comparison, whether an operation results in a negative value, etc.). Test instructions may be combined with different types of jump instruction which will transfer the control of the program to a different location. (A jump may be conditional on the result of a test or may be unconditional, in which case control will automatically pass to the given instruction such as the GOTO statement available in some languages.)

Input and output devices may be directly accessible to the processor, in which case a set of input/output instructions is needed to send the data to the desired device. This is not the only manner in which devices can be accessed; they may also be memory mapped, which means they are assigned one or more addresses and treated in the same manner as any other memory address, with the data being input or output instead of read or written to. Processor direct input and output can usually be accomplished by a shorter instruction, which will take less time to execute. This may be an important consideration in some processor applications.

Control instructions can be used to supply synchronisation signals and to suspend or interrupt the execution of a program.

Programmers using machine code may need to rewrite a module in order to make it execute as quickly as possible; this process is known as optimisation. It takes a great deal of skill and experience for a programmer to be able to optimise a subprogram, but it is an important concept, especially if that sub-module needs to be executed a large number of times.

Machine code is a processor-dependent method of programming, and as such has limited uses where a program is to be used on different computer platforms. However, this can also be an advantage, as an application written in machine code executes much more rapidly than one that needs translation. A second and more serious weakness is that a program written in machine code is designed to reside in one location only within the computer's memory. Thus, if the need arises for the program to be moved within memory (for example if it is made a submodule of a larger program), the addresses may need to be recalculated. To overcome this problem, symbolic assembly languages were developed and became the second generation.

Second-generation (symbolic assembly) languages

Symbolic assembly languages are not too far removed from machine code in that programs are developed by using the processor's instruction set; however, instead of using direct references to various memory locations, the programmer uses symbolic addresses. For the example above (adding *MARK* to *TOTAL*), the symbolic assembly language instruction may be:

```
CLA ADD MARK
```

where *MARK* represents the location in memory where the value of *mark* is stored. When the physical location of variables and instructions needs to be changed, the assembler takes care of the new memory locations.

The instructions available to an assembly language programmer are basically the same as those available to the machine code programmer, though some of the tedious work such as calculating relative addresses for jumps (so that the control passes to the correct instruction when the program is relocated to a different area in memory) is performed by the assembler. Many assemblers will allow internal program documentation in the form of comments, thus allowing the programmer to make notes in the source code of the assembler. This assists in following the logic in case of modification at a later date.

An assembler may also include a debugger which can be used to set breakpoints within the program so that, during the testing stage, the contents of various registers and memory locations can be viewed to see whether they contain predicted results when test data is used in the program.

Assembly language is much easier to handle than machine code for processors that have a large word size and vast amounts of primary storage. Some assemblers will attempt to optimise the code as it is translated into machine code.

Assembly language is preferred by a number of programmers since it is close to machine language in syntax and concepts, and runs faster if written well. Among this group would be those developing interfaces to various devices, creating ROM modules or developing microprocessor-controlled machines such as compact disc players, sewing machines, car fuel-injection systems and personal computers.

The following Z-80 assembly language code takes two values from two different addresses, adds them in the accumulator and then places the result in a third location.

```
LD A (NUM1)    LOAD THE VALUE FROM ADDRESS NUM1 INTO THE ACCUMULATOR
LD HL (NUM2)   LOAD ADDRESS OF NUM2 INTO THE HL REGISTER PAIR
ADD A (HL)     ADD CONTENTS OF MEMORY LOCATION STORED IN HL TC
                  ACCUMULATOR
LD (NUM3), A   PLACE CONTENTS OF ACCUMULATOR IN LOCATION NUM3
```

This code is equivalent to the BASIC code *LET* C = *A* + *B* or the Pascal code C*:* = *A* + *B*;

Assembly languages are still not easy to learn for the average computer user, as they are full of mnemonics. The range of commands and operations is limited to those that can be understood by the processor, and so even the simple processes of input and output can take a large amount of coding. With these problems in mind, third-generation languages were developed. (See Figure 5.22 on page 145 and Figure 5.23 on page 146.)

Exercise 9.1

1. Name the two methods by which first-generation computers were programmed.
2. Why do we use binary, hexadecimal and octal in computers? Count to 32 (decimal) in hexadecimal, binary and octal.
3. Name the five categories of machine instruction.
4. Describe the process of optimisation.
5. Describe one advantage and one disadvantage of first-generation languages.
6. How are programs developed in symbolic assembly languages?
7. What is a debugger used for?
8. Why were second-generation languages found to be inadequate, forcing the development of third-generation languages.
9. What is a jump in assembly language?
10. Find some uses of assembly languages in today's society.

EXTENSION ACTIVITIES

11. Research and find a short program written in assembly language. Explain its purpose line by line.
12. Research the methods by which ROM modules are created and 'burnt' onto ROM chips for microprocessor-controlled machines.
13. Obtain examples of internal documentation from an assembler program.
14. Write a short presentation on the life of Alan Turing and his importance to the progress of computing.

Third-generation languages

The third generation of languages marks a great leap forward in program design, since the instructions available to the programmer are closer to the types of symbolic languages used in science (for example FORTRAN) or business (for example COBOL). These languages were the first to be known as high-level languages. The most significant advantage of third-generation languages is that they are independent of the computer system. A single program, written in FORTRAN for example, could be compiled and run on a number of different computer types provided the compiler could convert the source code into the appropriate object code. As computer technology improved, it became possible for some third-generation languages to be interpreted line by line. (Many of the earlier personal computers such as the Apple II and the Commodore 64 were equipped with BASIC language interpreters which brought sophisticated programming capabilities to the home-user for the first time.)

Our simple instruction (adding *MARK* to *TOTAL*) in a language such as BASIC would become:

```
TOTAL = 0
TOTAL = TOTAL + MARK
```

The types of application being written influenced the development of third-generation languages. FORTRAN and ALGOL were aimed at the scientific and mathematical users, so they were constructed to allow easy translation of formulas into code, input data and output data in forms that were appropriate to the sciences. They were also equipped with a library of appropriate modules, which could be called very easily. (For example, a square root module may be called by the coded line SQRT(X) where X is some value or variable name.) Business has different needs from those of mathematics and science and so languages were devised, the best known being COBOL, to include common business notations and modules. (For example, to calculate the interest on an amount A, at a rate R%, over a period of time T may be called by the coded line INTEREST(A,R,T), the compiler taking care of the processes of passing the values to the sub-module and back to the main program, and the actual calculation involved.)

Third-generation languages were developed for specific purposes, not because it was desirable but because the technology available for compilation could not handle a true general-purpose language. Compilation involves the conversion of the high-level language into low-level language, and may also attempt to optimise the code so that the program runs more efficiently. Languages were also developed so that each of the reserved words differed in their combinations of first and third letters. The compiler would look first at these letters, thus speeding up the compilation of the code.

A genuine attempt to standardise the construction was made with many third-generation languages, but different manufacturers offered slightly different versions of the languages that took advantage of features available on their computer. The effect of this was to make the programs less portable (the ability of a program to be run on more than one computer platform) than was desired by the original developers of the language. Attempts were also made to create minimum-standard versions of languages which provided a common base on which various implementations of the language could be made by developers.

For example, ANSI (American National Standards Institute) minimal BASIC provides a base on which many versions of the language have been developed. This means that common instructions, such as those used for input and output, are implemented in all versions, but some specialised instructions, for example those involved with graphics display on a VDU, may be implemented in different ways for different versions. A minimum-standard version of a language allows programmers to create applications that will execute properly on computers with many different processors or operating systems.

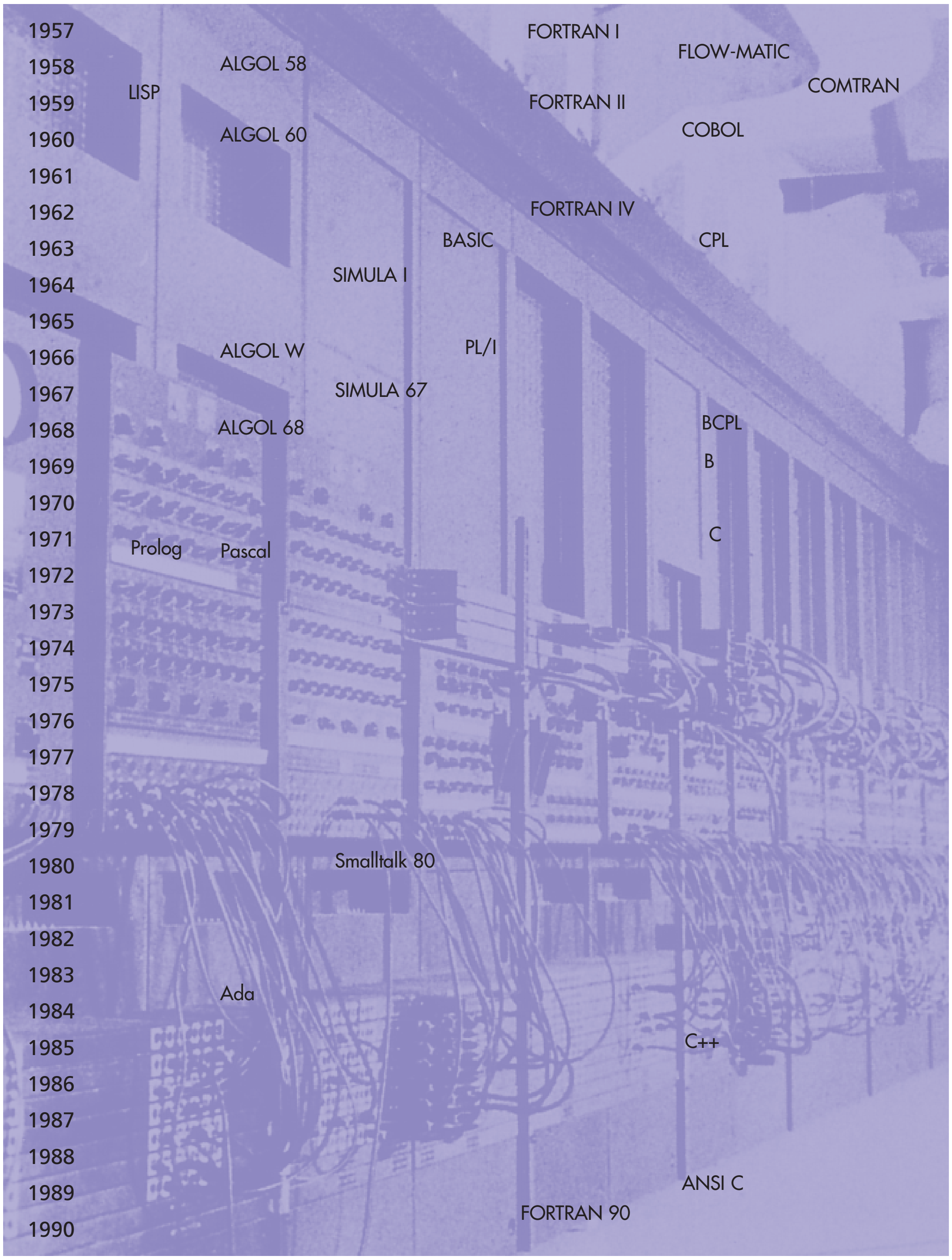


Figure 9.2 Major language development time-line.

Fourth-generation languages

Fourth-generation languages are distinguishable from the third generation in that they allow programs to be expressed more in terms of *what* the program has to do rather than *how* it is to be done. Up to the third generation, computer technology shaped the manner in which programs were developed, as it was the most expensive part of the system. With the power of the computer increasing and its cost decreasing, the human aspect has become more expensive than the computing aspect. Languages developed in the fourth generation have attempted to make interaction with the human easier. However, the computer system needed to support these languages has become more complex in order to cope with them.

Fourth-generation languages are characterised by their similarity to human speech or thought; simple programs in languages such as Logo and Hypertalk are very easy to understand and construct. Complex procedures such as creating music or animations can be accomplished easily even by people with very little programming experience.

Using a simple example, a fourth-generation language may appear thus:

```
USE pupils
SUM mark1
TO total
```

LISP (a **list** processing language) is a language that sits on the border between the fourth and fifth generations. It is favoured by those involved with artificial intelligence applications.

A LISP program which finds the percentage change from A to B would be written in the following manner:

```
(DEFINE (PC B A) (QUOTIENT (TIMES (DIFFERENCE B A) 100) A))
```

Notice that the operations (*QUOTIENT*, *TIMES* and *DIFFERENCE* for this example) in LISP are stated before the operands (*B* and *A* above). This notation is called reversed polish notation, or RPN, and is similar to the one used on many of the early scientific calculators. In RPN, adding the two numbers 5 and 23 would be keyed in a manner similar to:

$\boxed{+}$ $\boxed{5}$ $\boxed{\text{ENTER}}$ $\boxed{23}$ $\boxed{\text{ENTER}}$

A program in LISP is really only a list of functions and arguments enclosed in brackets. The placement of the brackets determines the way in which the list is processed. LISP lends itself to artificial intelligence applications since variables may be set to non-numerical values just as easily as numerical ones.

LOGO is a high-level language developed from LISP by Seymour Papert especially to introduce very young children to computer programming. The most widely used subset of the LOGO language is *Turtle graphics*, whereby children create drawings by 'driving' a cursor called a turtle around the computer screen by means of a list of instructions. (In the original application of Turtle graphics, the turtle was an electronic device, connected to a mainframe computer, which held a pen and was programmed to draw on sheets of paper; consequently, some of the instructions in LOGO were directly related to the physical turtle, for example the PENUP and PENDOWN commands.) An example of a typical LOGO module would be one to draw a hexagon on the screen:

```
TO HEXAGON . SIZE
REPEAT 6[FORWARD SIZE RIGHT 60]
END
```

The first line declares a module called HEXAGON, which takes an input called SIZE. The second calls on several modules that are built into LOGO (called library routines) to repeat the steps of drawing the required length interval and a 60° right turn a total of six times in order to construct a regular hexagon on the screen.

Fifth-generation languages

Fifth-generation languages use artificial intelligence methods to improve the interaction between the human and the computer for problem solving. These languages use the declarative programming paradigm, with the emphasis being placed on an approach known as logic programming. Declarative programming concentrates greatly on the desired outcome of the process rather than on the manner in which it is to be achieved. Many fifth-generation languages employ problem-solving algorithms that have been based on the principles of formal logic.

Languages such as PROLOG employ a series of initial statements which form the basis for a process called resolution. Resolution involves using the initial statements to deduce the truth of a further statement. An example of resolution is as follows.

Suppose that we know that Mary is either working or sick. If we are then told that she is not sick, we can deduce that she is at work. The two initial statements are that 'Mary is at work' and 'Mary is sick'. The resolvent statement is the deduced one that since Mary is not sick she is at work.

The process of resolution mirrors the way in which we come to a conclusion when confronted by a series of statements.

Exercise 9.2

1 Answer the following questions by filling in the space provided.

- a The development of software is closely associated with the development of new ________.
- b Computer languages can be categorised into ________ generations.
- c First- and second-generation languages are ________ dependent.
- d Second-generation languages replaced the sequences of binary digits with ________ codes.
- e Fourth-generation languages are more concerned with what needs to be done than ________ it is to be done.
- f The last generation of languages allows the programmer to code complex knowledge which the computer can draw ________ from.
- g Data processing operations fall into one of the five categories of arithmetic operations, bit manipulation, increment/decrement, shift/skew operations and ________ ________.
- h An example of a third-generation language is ________.
- i ________ involves the conversion of the high-level language into a low-level language, and may also attempt to optimise the code so that the program runs more efficiently.
- j ________ are characterised by their similarity to human speech or thought.

2 Identify each of the following code fragments as first, second, third, fourth or fifth generation. Give reasons for your choices.

- a =Sum (a9..b21)
- b A2 B4 32 45 EF FF 02
- c REPEAT 12[PENDOWN FORWARD 10 PENUP FORWARD 5]
- d LDA, VALUE
- e REPEAT X:= X + 1 UNTIL X > 20

3 Describe the importance of third-generation languages.

4 Explain the meaning of the term 'compilation'. Describe one method of speeding up compilation used in third-generation languages.

5 What is portability of programs? Why was portability difficult to implement in third-generation languages?

6 What is the minimum-standard version of a language? Describe the role of ANSI in the production of minimum-standard versions.

7 Describe the differences between third- and fourth-generation languages. Give code samples to help illustrate your answer.

8 What is the meaning of the statement 'the human aspect has become more expensive than the computing aspect' with reference to fourth-generation languages?

9 Design and implement a procedure to draw a street full of houses using a version of Logo.

10 Describe the differences between fourth-generation languages and fifth-generation languages. Where appropriate, use code samples to illustrate these differences.

11 What does the term 'resolution' mean when applied to fifth-generation languages?

12 Explain the differences between high-level and low-level languages. Include the terms 'source code' and 'object code' in your answer.

EXTENSION QUESTIONS

13 Research the development of PLANKALKUL written by Konrad Zuse. Why was this programming language a major breakthrough? Present your answer as a short article written as a historical review.

14 Grace Hopper was an important figure in the development of high-level languages. Investigate and report on the contributions made by Grace to computer programming.

15 Write a short presentation on the development of FORTRAN, ALGOL or COBOL. Include in this presentation a listing of a program written in your chosen language.

16 Describe the development of the programming language Ada.

17 Explain the purpose behind the development of Pascal as a programming language. Explain how the structure of the language achieves this purpose.

Paradigm specific concepts

A large number of computer programs are written in a paradigm or model in which algorithms are expressed as a sequence of commands. These commands are followed line by line, sequence by sequence until the instructions have been completed.

This paradigm of a program is based on the Von Neumann computer architecture model that, in turn, is based on the model of computation put forward by Alan Turing.

The imperative programming paradigm is easiest to implement on the Von Neumann computer. The basis of the Von Neumann computer is the separation of the processing components from memory. In contrast, the human brain operates in a manner that combines memory and processing. In the Von Neumann computer, the storage of instructions and data is in the same memory locations, and processes within the processor govern whether the value stored in a particular location refers to instructions or data. This architecture lends itself to imperative languages, which employ variables to model memory locations and assignments, which model the processes of the transfer of data from location to location. When branching and iteration (the process of repetition of a number of steps) are added to the language structure, a versatile and powerful language is produced.

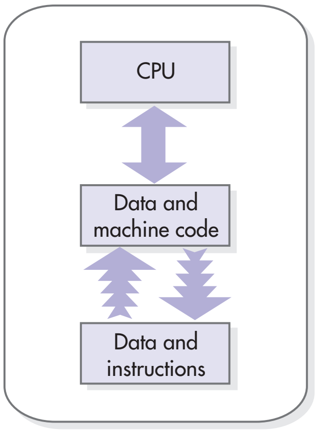


Figure 9.3 A block diagram of the Von Neumann architecture.

An introduction to object-oriented languages

Object-oriented or event-driven paradigm programs are constructed from objects and messages. Objects can be thought of as a 'black box' which sends and receives messages. The object may not be understood by the programmer but can still be used effectively. (See Figure 9.4.)

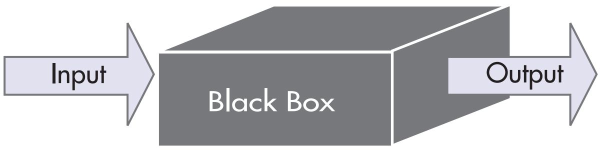


Figure 9.4 A 'black box' can be used to represent a process where we have no interest in the way that it works.

The statement ‘pick up the inkbottle’ can be used to demonstrate this. In procedural language the algorithm would involve a step-by-step solution explaining every part of the program. It could end up being written as follows.

```
BEGIN
    move arm in an arc until it comes to the inkbottle,
    open hand keeping fingers apart,
    place around inkbottle,
    close hand
    gently and firmly grasp inkbottle
    move arm upwards keeping inkbottle as still as possible.
END
```

Whereas procedural languages specify a step by step account, object-oriented languages are concerned with the object. These objects are responsible for responding to the requests from the clients. The way in which they respond to these requests is up to them. (Thus the responder to the statement ‘pick up the inkbottle’ could pick it up between their teeth or even by their toes.)

There is obviously a lot going on behind the scenes, but this is the basic concept of object-oriented languages—the objects are responsible for their own actions.

Until the advent of object-oriented languages, models were created that kept the code separate from the data. In object-oriented languages the code or function and the data are kept together in this object. There is no need to see what is inside the object; it only needs to be manipulated; hence the reference to the ‘black box’.

A class of numbers called counting numbers (0, 1, 2, 3, 4 …) can be created with the properties that they are ordered and represent quantities. Associated with this class could be the operations of addition and multiplication. A subclass can then be created of the counting numbers consisting of 0 and 1. This subclass automatically inherits the properties of order and representation of quantities and also the operations of addition and multiplication, as they are defined for the larger class. However, binary addition and binary multiplication may also be defined for the subclass.

This shows the way in which objects in an object-oriented language are divided into classes and subclasses, with subclasses inheriting the properties of their parents. Objects receive messages and the way they deal with these messages is called ‘methods’.

Object-oriented programming is ideal for writing the complex programs needed for computer-controlled systems, as an object can represent each component of the system.

From the programmer’s point of view, object-oriented languages have many advantages. A programmer can lift a piece of code from an original program and by manipu lation and by gaining new characteristics is able to carry out new tasks within the program. Programmers are able to do this by knowing how to manipulate the object—not how the object works. The programmer is able to save time, money and effort merely by performing this manipulation. The only disadvantage is that if an object is deleted any connections to the object are redirected correctly. Remembering this, revision or even rewriting can easily proceed.

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

Figure 9.5 Binary addition and binary multiplication defined for the subclass (0, 1) of the counting numbers.

There are two key notions in understanding object-oriented languages: inheritance and encapsulation. Associated with these two basic concepts are methods, classes, abstraction and polymorphism.

Encapsulation

When programmers embark on a project that has grown in size beyond a few thousand lines, they face a variety of problems.

If they create a group of subprograms or modules it can be difficult to follow the project's structure (similar to a large relational database). How are the sections joined together? How do the parameters relate to each other and what are the links between the components? Compilation of such a program can become costly in both time and effort.

One method used to save this effort is encapsulation. The programmer has the ability to group subprograms and related data either individually or in groups of components that are related.

These groups of subprograms may then be compiled independently or left alone, depending on what the programmer wants. Compilation of some sections of the program for use as general components will save time and effort in later projects.

As programming languages have evolved, the techniques of encapsulation have evolved, with different languages containing subprograms, providing library routines and simplifying the interfaces among encapsulated objects.

Polymorphism

Polymorphism is a powerful concept that greatly reduces the effort required to extend an object-oriented model during the development and maintenance stages.

To understand polymorphism, consider a program to draw six cars. If this were being done in a traditional program, a case structure would be created and modules drawn for each car part.

If we wished to expand this program, we would need to change not only the case structure but the control program as well.

The difference with polymorphism is that the class structure comes into play. The cars would become a subclass of the general class called 'vehicle', which would define the draw operation. To add a new car to the list, we would simply add a new member to the subclass 'cars'.

The subclass inherits all of the properties of the parent class 'vehicle', and to draw one of the original cars, or an additional member of the class, the appropriate command would be sent to the required object.

The basic concept of polymorphism in object-oriented languages is that for effective maintenance or extension of a program only the addition of a subclass is required; the control program does not have to be adapted.

Inheritance

It became increasingly obvious to programmers in the late 1980s that they were rewriting pieces of code that were similar to previously written code; their previous work was unable to be utilised effectively. The problem was that the previously written code was only a partial answer to the required solution rather than being the whole answer.

'Inheritance' is a solution to this problem. In addition, it helps with the organisation of the program. Quite simply, inheritance in a hierarchical structure is the ability of the client to have all of the characteristics of the parent and to gain new ones.

Inheritance within a class can be via a single parent (referred to as single inheritance) or via more than one parent (referred to as multiple inheritance). Thus, in inheritance a class can have inherited characteristics from a parent or multiple parents. The class also has the ability to gain new characteristics that enable it to be adapted to new requirements.

This simple idea has enabled programmers to effectively reuse previously written code and to utilise work done at an earlier time.

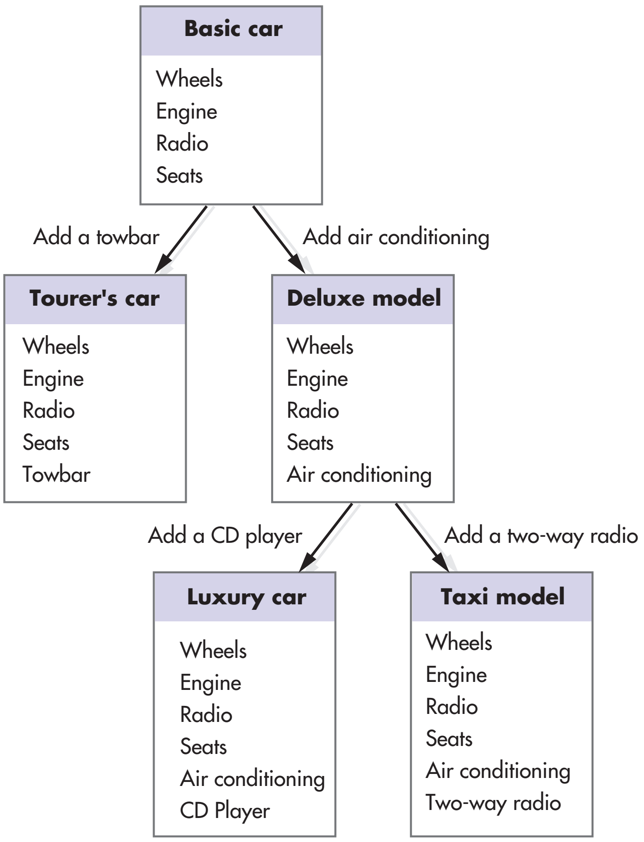


Figure 9.6 A subclass inherits all the properties of the class to which it belongs.

Classes

Abstract data types in object-oriented languages are called classes, remembering that data abstractions refer to the attributes or properties of that data item.

A class generally refers to a group or collection of similar objects. The elements or class instances are called objects.

A class defined through inheritance from another class is called a subclass. The class from which the subclass is derived is called the parent class or superclass.

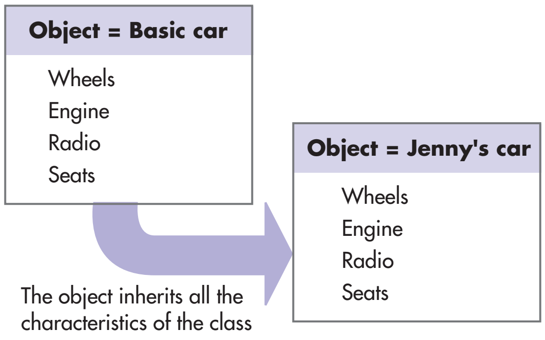


Figure 9.7 An object inherits all the characteristics of the class to which it belongs.

Methods

Subprograms that define the operations on objects of a class are called methods; they may also be referred to as operations or services. In object-oriented languages an object encapsulates data represented by a collection (or a list) of attributes together with the algorithms that are used to process them.

In object-oriented programming the term 'message' is used to refer to the means by which objects interact.

When an object receives a message from a source object, it responds in the appropriate manner, passing its response on as a further message. The output message may contain the result of an operation or it may provide information to the source object that an action has taken place. For example, a message may be sent to a circle-drawing object to draw a circle of radius 20 pixels with its centre at the point (100, 150) on the screen. The object will 'draw' the circle by making the necessary calculations and place the representation of the image in the video memory of the computer. A message will then be sent to the source module 'saying' that the circle has been drawn.

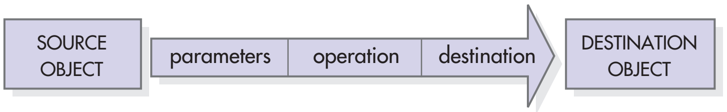


Figure 9.8 A message carries three vital pieces of information: its destination, an operation and the parameters.

Messages must have three specific parts: the destination object, an operation that has to be performed, and the parameters to be used by the destination object in performing that operation. In the circle-drawing example, the radius of the circle and the coordinates of its centre are the parameters that had to be sent and the operation was to 'draw' the circle. The circle object may also contain a method for erasing a circle.

Abstraction

Abstraction is extremely important because it allows the programmer to simplify the programming process.

Abstraction in a modular solution to a problem can be viewed in levels. The solution is stated in broad terms using the desired programming language. The language itself allows the programmer to generalise when creating the solution rather than concentrate on specific details. By using abstraction, a problem can be viewed in familiar terms. For example, when we view a screen of word-processed text, it appears to us as words written on a sheet of paper. We manipulate the text as if it were on paper; this is an abstraction. The actual processes involved in manipulating these words and the ways in which they work are completely hidden.

By being able to concentrate on the more essential parts of the program and, to some extent, ignore the less important, the programmer has an easier path to the problem solution.

There are two essential forms of abstraction:

- data abstraction, which is looking at data items in familiar terms such as characters and numbers
- process abstraction, which is a sequence of instructions that have a clearly defined and limited purpose.

Description and history of languages

C++

C++ was developed from languages that were originally created at Cambridge University during the early 1960s.

C was the name given to the in-house language called B (BCOL – Basic Combined Programming Language, which itself was a descendant of ALGOL 60). C was designed and implemented by Dennis Ritchie at Bell Laboratories in 1972. In 1980 Bjarne Stroustrup, also of Bell Laboratories, modified C. Further modifications were made through the 1980s until Release 2.0 was made available in 1989.

C++ is a hybrid language which reflects its origins together with the influences of the culture in which it was created. C++ is a very popular language for a variety of reasons, including:

- There are excellent compilers available.
- C++ is seen as the flagship of object-oriented programming methodology.
- C++ is backward compatible with programs written in C; that is, programs written in C can be compiled in C++.

The influences of previous languages can be seen in the hybrid C++. C was the starting point, and the influences and improvements that came from the language Smalltalk saw C evolve to become the 'flagship' of object-oriented programming.

C++ is a compiled language, which means that the computer translates the code into an intermediate form called an object file. The linker is then invoked to link the object file with any required libraries and convert it into an executable program. A compiled language is in general more complex, the advantage being that a compiled language is much faster than an interpreted language.

The negative aspect of C++ is that, as it is a hybrid language, it has inherited many of the faults of C and it is perceived by many to be too large and complex a language.

Java

Java is a programming language devised at Sun Microsystems in 1990. It was originally designed for use on small consumer electronic devices.

The experts at Sun reviewed the two major languages of the time, C and C++, and decided that neither met their requirements for an extremely reliable yet simple language. None of the devices that Java was designed for made it to the marketplace.

Java was originally developed from C++ but with the central belief that a simpler language that was also dependable was needed.

The developers at Sun created a system whereby, instead of compiling a program specific to one operating system, they developed a programming language that had portability or, as Sun likes to call it, 'write once, run anywhere'.

By 1996 the two factors that were making Java popular were that 'write once, run anywhere' had become very popular with Web developers and Java was seen by many to be an alternative to Microsoft products.

The issue of portability is very important to Java. This issue has been solved with the 'Java virtual machine' whereby, if an applet (a small program that runs on the web client) is called, the applet is downloaded from the web source and interpreted on the machine. The output is then displayed.

From a programmer's point of view the code can be written once without worrying about the specific operating system it is to operate under. The program can be used on any computer that is running an appropriate interpreter.

Is there a price to pay for this portability? Yes, and the price is speed. If an application is going to run anywhere, the translation work being done by the compiler cannot be done in advance but rather as the program is being executed.

What does the future hold for Java? Will it be only for the Web? Sun Microsystems is obviously aiming at expanding the usage of Java to applications apart from the Web and has claimed that worldwide there are over 40 000 Java programmers creating a wide range of applications, ranging from the Web to server and desktop applications.

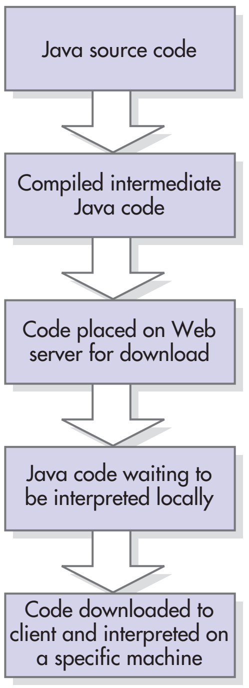

Java source code
Compiled intermediate Java code
Code placed on Web server for download
Java code waiting to be interpreted locally
Code downloaded to client and interpreted on a specific machine

Figure 9.9 A Java applet is portable, since its code is interpreted when executed on a machine.

A sample piece of Java code showing the IF/THEN statement would be:

```
If ( age == 18)
 {
 System.out.println('You are just old enough to vote!')
 }
else if (age < 18)
 {
 System.out.println ('You are too young to vote!')
 }
else
 {
 System.out.println ('You're old enough to vote!')
 }
```

Declaration of a variable is performed by a simple statement such as:

```
int age = 30;
```

These examples show how easy Java code is to read.

Delphi

Delphi, like the previous languages, was derived from another language. The influences of Pascal on Delphi can be clearly seen. The programmers at Borland who created Delphi were responsible for this influence.

Delphi is seen mainly as a development tool for Microsoft Windows applications, being used as a tool to create programs with a graphical user interface (GUI).

Delphi is also referred to as object-oriented Pascal, as it contains many of the elegant and inelegant features of Pascal.

Programming in Delphi involves a combination of objects or components, events, tools and controls.

Controls are those components that the user can manipulate, for example buttons, edit boxes, labels and option buttons. All of these objects are standard Windows elements. An event occurs as a result of a component's interaction with the user. Windows and tools are the common parts of the Windows GUI and can be added to the program in a similar way to adding an object in a drawing program.

Program creation is quite simple. You first create the window the user will see by placing on the screen the components that the user will manipulate. The next stage is to decide what will happen when the user interacts with each component. Finally the procedures are designed. These procedures specify the events that take place once the user has made a decision.

Borland has developed visual component libraries to enable ease of use for the programmer. These libraries include classes of objects such as windows, buttons and custom tools as well as non-visual lists to help with event procedures. Borland has also developed a means by which Delphi, with created reports and forms, can access local and network databases.

Delphi is thus a powerful tool which enables the programmer to design the interface and implement an event by the simple click of a mouse on a specified section of the window.

The sample code below responds to a given event. The message 'Hello World' will appear when the button is pressed.

```
Procedure Tform.Button3Click(Sender :Tobject);
Begin
Edit1.text := ('Hello World')
close
end
```

The first line of the procedure tells Delphi that the button is part of the form and that it will respond to a mouse click. The action of the mouse click will display 'Hello World' on the screen, then the program will close. The body of the program is held between the standard constructs 'Begin' and 'end'.

It is interesting to compare the three different ways a print statement is written. The syntax used is:

```
Pascal:     write ('hello world');
QBASIC:     PRINT 'hello world'
Delphi:     edit1.text := ('hello world');
```

The purpose of each of the code samples is easily understood; some of the samples can even be understood by people with little or no programming experience.

Exercise 9.3

1 Fill the blank spaces with the appropriate term from this list:

class, encapsulation, inheritance, polymorphism, process

- a ________ is the ability of the client to have all of the characteristics of the parent and gain new ones.
- b A ________ generally refers to a group or collection of similar objects.
- c ________ abstraction is a sequence of instructions that have a clearly defined and limited purpose.
- d ________ leads to effective maintenance or extension of a program. You need only add a subclass rather than having to adapt the control program.
- e ________ is when a combination of data and objects is made invisible to other objects.

2 Describe polymorphism and explain why it is an important feature of object-oriented languages.

3 Explain why we can use the reference to a 'black box' when referring to objects in object-oriented languages.

4 In your own words write a list of 10 instructions you would expect to see in a program that tells a robot to pick up a jug of milk and pour some of it into a cup or mug.

5 Why is inheritance important in helping the programmer save time in object-oriented languages?

6 Describe the term 'abstraction' and explain its importance in object-oriented languages.

7 Research and find some code in C++ that will declare a simple variable and print the same variable to the screen.

8 Investigate the influence of Phillip Kahn on computer technology and programming. Explain why he would be referred to as a computer entrepreneur.

Logic paradigm

The logic paradigm is a totally different kind of programming paradigm. The languages used for logic programming have been called non-procedural, declarative or logic. Logic programming is associated very closely with the artificial intelligence world. The main thrust of research into this style of programming is to look for ways to make computers do things that human beings do better at the moment. These include the ability to learn, to process knowledge, to make inferences and to gain knowledge. From these processes human beings can provide a solution to a problem.

With a conventional programming paradigm a particular set of statements will be executed in a rapid and precise order to produce a result. In the logic paradigm a detailed description of a set of statements that describe what is true about a desired result is provided to the computer, which can then infer a result.

This has been interpreted to mean that the final algorithm reflects the independence of logic and control. In this paradigm logic specifies what is to be solved and control specifies how it should be solved.

For a logic programming language to be effective there must be a clear way of supplying the computer with the pertinent information as well as the methodology to, in some way, solve the problem.

In logic programming there are only three types of statements: facts, queries and rules.

How does logic programming work in theory?

The programmer writes a 'database' of facts, for example:

Wet (rain) which means 'Rain is wet'

together with 'rules' such as

Mortal (x) : - human (x) meaning 'x is mortal is implied by x is human'.

Facts and rules are together known as 'clauses'. The concept is based on **Horn clauses**. Simply put, clauses are equivalent to facts and **Horn clauses** can be used to express most logical facts. Consider the following:

a_1 or a_2 or a_3 … or $a_n \rightarrow b$

The section pertaining to a is called the body of the clause and that pertaining to b is called the head of the clause, and this implies that if a_1 or a_2 or a_3 or … a_n is true, so is b.

Another way of looking at this language paradigm is that it is based on the logic of equations and functions. For example:

If $A(g) = B(h)$ and $B(h)$ = house then $A(g)$ = *house*.

The interface of a logic program with a user and what goes on behind the scenes would follow a similar line to the following.

The user supplies a goal, which the system attempts to prove using resolution or **backward chaining**. This involves matching the current goal against each fact. If a goal matches a fact, the goal succeeds and the resolution proceeds; if it matches a rule with all of the sub-goals, the rule succeeds.

The point at which a possible clause is chosen is known as a branch; if a subsequent choice fails, control returns to this point—the last point that was true—and it continues through a different clause set.

The user is then informed of the success or failure of the choi ces made and the variables that were input to get to that point. A solution is also presented.

To enable this process to occur there are powerful computations going on behind the scenes, involving heuristics, goals, inference engines and backward and forward chaining.

Heuristics

Heuristics involves finding a set of rules or a procedure that finds satisfactory solutions to a specific problem.

The heuristic approach enables the generation of many different solutions from which a decision can be made, each of them being worthy in some way. The optimal solution may not be found, but a solution of some worth will be.

Goals

When a group of rules or facts from the database are able to be matched, the goal has been achieved. For example, you may think, through observation, that the Earth is flat. This is your original proposition, or hypothesis. If you were able to prove, from a number of different facts, that the Earth was indeed flat, you would have achieved the goal.

Inference engine

An **inference engine** is the processing portion of an expert system. With information from the knowledge base, the inference engine provides the reasoning ability that derives influences or conclusions on which the expert system acts.

Backward chaining

Backward chaining is a problem-solving procedure that starts with a statement and a set of rules leading to the statement and then works backward, matching the rules with the information from a database of facts until the statement can be proved either right or wrong. This method works better with a small set of results.

In backward chaining the result on the right-hand side of the diagram (see Figure 9.10) is the beginning of the process of assessing the results.

The methodology used is that the program works backward from completion to rule, collecting facts until able to continue to another rule, until eventually the source of the material that can be hypothesised is reached. The basis of backward chaining is that a source can be found without the need to obtain all the data.

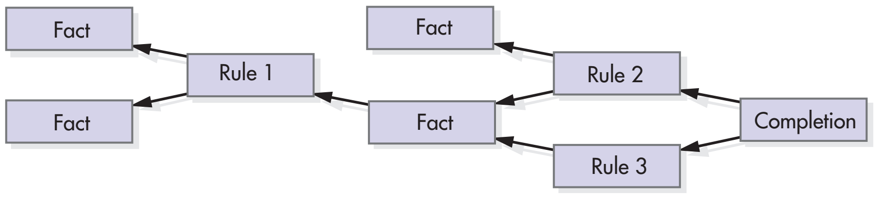


Figure 9.10 Backward chaining starts with the statements and rules, then works back to a solution.

Forward chaining

Forward chaining is a problem-solving procedure that starts with a set of rules and a database of facts and works to a conclusion based on facts that match all the premises set forth in the rules. This methodology works better when there is a large number of correct results.

Expert system shells

To understand the concept of expert system shells we must first understand what an expert system is. An expert system is a program that is designed to act like an expert in some narrow field or application. An expert system has the ability to solve problems that require expert knowledge, so they have also been called 'knowledge-based systems'.

Conventional computer programs solve problems and perform their designed functions by using conventional decision-making logic. The basic algorithm is designed to solve a specific problem and is designed to deal with any uncertainty such as boundary conditions and unclear data. If the knowledge or requirements change, then the program has to be changed and rewritten.

An expert system program is designed to deal with uncertainty within its problem-solving function. The logic of the expert system has a means to deal with this uncertainty. The user interface is usually very simple and clear and able to explain its actions to the user.

The expert system consists of three main sections: a knowledge base, an inference engine and a user interface.

The **knowledge base** usually consists of a database built up by a knowledge engineer that is specific to the application being worked with and includes simple facts, rules that describe relationships and possible methods. This database must also have the ability to grow as it learns new facts, and is therefore dynamic.

An **inference engine** is the control engine; this applies the knowledge to the data to arrive at some conclusion. Having the knowledge base alone would be useless; there needs to be a way of manipulating the knowledge to deduce something from it. As the knowledge base can be quite large, the inference engine searches through it using means such as forward and backward chaining.

The **user interface** provides the means for a smooth and clear communication between the user and the system. It should give an insight into the problem-solving process carried out by the inference engine.

The inference engine and the interface are seen as one module, which is usually called the expert system shell.

As Figure 9.11 shows, the shell is independent of the knowledge base, so it is theoretically possible to develop a shell that can be used universally and to simply place a new knowledge base for access to a particular expert. This is a very simple introduction to expert systems and shells.

As an example, an expert system could be constructed that contains the information required by a doctor in general practice to diagnose common diseases. The doctor would provide the expert system with the symptoms and the expert system would match these with the symptoms of various diseases. The likelihood of the patient suffering from each of the identified diseases could be conveyed to the doctor by the interface. The expert system could also suggest a suitable treatment for the disease.

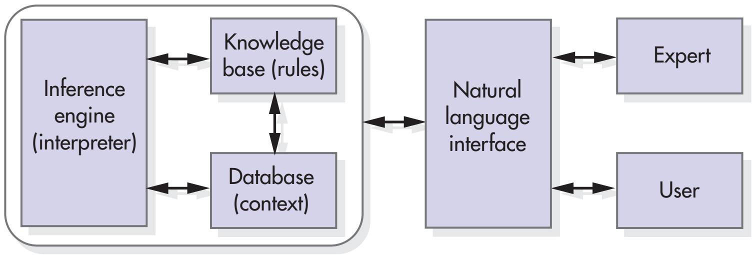


Figure 9.11 An expert system resembles a production system.

Category	Action of system	Type of system
Interpretation	Infers situation descriptions from sensor data	Speech recognition, image analysis, surveillance
Prediction	Infers likely consequences of given situations	Crop estimation, weather forecasting
Diagnosis	Infers system malfunctions from observations	Electronic, medical
Design	Builds objects given their description	Circuit layout, financial budgeting
Planning	Designs actions	Automatic programming, military planning
Monitoring	Compares observations in order to plan for failures in procedures or machines	Nuclear power plant control, fiscal management in stockmarkets
Debugging	Prescribes solutions to malfunctions	Computer software
Repair	Executes a plan to administer a prescribed solution or remedy	Automobile, computer
Instruction	Diagnoses and corrects student behaviour	Tutorial, remedial study
Control	Interprets, predicts, repairs and monitors systems operations	Air traffic control, war battle management

Table 9.2 Typical expert systems applications

Prolog

Prolog stands for **pro**gramming in **log**ic. It is an example of a language that has grown and evolved since the early 1970s.The early developers of Prolog included Robert Kowalski at the University of Edinburgh, Maarten Van Emden at Edinburgh and Alain Colmerauer at Marseilles.

Colmerauer, with others, developed the first logic programming language, Prolog. The specialised theorem prover that he developed embodied Kowalski's procedural interpretation; that is, languages following this paradigm have their commands executed in the order necessary to achieve a solution, not necessarily the order placed in the source code. In other words, the 'what' not the 'how' of the solution.

The journey that Prolog has travelled began with the two groups in Edinburgh and Marseilles working together, then drifting apart, and the development in the 1970s of efficient implementation by David Warren. There was little interest in Prolog and logic programming until the early 1980s when Japan announced plans to introduce a fifth generation of computer hardware, with the aim of building parallel knowledge-based machines that would accept natural language input and process large quantities of information. The kernel language for this would be Prolog.

Prolog has grown from this to be used for theorem proving, relational database design, software engineering, natural language processing, knowledge representation in artificial intelligence and expert systems programming.

Prolog is seen by many as the first step towards non-procedural programming, where the user and the programmer can concentrate more on what needs to be done than on how to do it.

Here is an example of Prolog where facts may be expressed in the following manner:

```
Male (james)
Female (jessica)
Likes (jessica, james)
Film (crocodile dundee, hogan, koslowski)
```

and rules in this way:

```
brother (a,b) := male (a), parent (a,z), parent (b,z)
```

This statement says that *a* is the brother of *b* if *a* is a male and if the parent of *a*, in this case *z*, is the same as the parent of *b*. Queries can also be expressed in this language in the following manner:

```
? – likes (jessica, james)
? – brother (james, jessica)
```

In order to draw the appropriate conclusions for the problem at hand, a Prolog system may need to process a number of rules. The programmer does not need to state the sequence in which those rules are processed, only the rules needed. The software is then able to determine the rule needed at each stage of processing.

Exercise 9.4

1 Fill in the spaces with the most appropriate word from the following list:
rules, desired result, functions, user, declarative, knowledge, queries, equations, inference

- a Another name for the logic paradigm languages is a __________ language.
- b In the logic paradigm a detailed description of a set of statements that describe what is true about a __________ is given.
- c In logic programming there are only three types of statements: __________, __________ and __________.
- d The logic paradigm is based on a logic of __________ and __________.
- e The expert system has three main sections: a __________ base, an __________ engine and a __________ interface.

2 Explain how backward chaining is able to achieve a hypothesis.
3 Complete the following line that shows the logic of an equation:
if hose(red) = x(Y) and x(Y) = plane. What is the value of hose(red)?
4 Explain the meaning of the term 'goal' when used in logic programming.
5 What does it mean when a database is said to be dynamic?
6 Explain the workings of an expert system shell with the aid of a diagram.
7 Describe the purpose of an inference engine.

EXTENSION EXERCISES

8 Niklaus Wirth and Robert Kowalski were closely associated with the creation of Prolog. Investigate their lives and the influences they have had on the creation of Prolog.
9 What is the fifth generation of computer hardware?
10 Describe the purpose of a kernel language.
11 Why did the two groups who developed Prolog in Marseilles and Edinburgh drift apart after the language creation?
12 Explain why Prolog was developed in Europe rather than in America.

Functional programming

Functional programming is so called because a program consists entirely of functions. There are many terms that are used when referring to the functional paradigm: functions, side effects, referential transparency, recursion. A simple explanation of these will suffice at this level.

Functional programs contain no assignment statements, so variables, once given a value, never change. Functional languages forbid assignment to global variables. Each expression is a constant. Thus, the result of a function will, under all conditions, be determined solely by the value of its arguments. This ensures that there are no side effects from these expressions.

A side effect occurs simply when the function changes either one of its parameters or a global variable such as can be done in a simple assignment statement. Because a functional language cannot have variables, they do not have loops as is seen in structured programming; rather, they use recursion where a function or procedure calls itself.

For example, we could define a counting number as being one more than the previous counting number. However, we need a starting point for this definition to hold, so we define the first counting number as being 0. This means that the next counting number is one more than zero, and so all the other counting numbers can be defined. You will notice that the term 'counting number' is used in its own definition. This process of using an item to define itself is called 'recursion'. As already seen, a starting point must also be given to a recursion.

The basis of the paradigm is the function. A function from mathematics is a rule of correspondence that associates to each member of its domain a unique member in the range. Functions can be either simple assignment statements or joined together to make more complex ones.

The key to functional programming's power is that it allows greatly improved modularisation. This is also the goal for which functional programmers must strive—smaller and simpler and more general modules, glued together to make more complex functions and rules.

Functional programming languages are being used more frequently, but they still have two main drawbacks: efficiency and suitability for applications with a strongly imperative nature.

Until recently, programs written in a functional language ran more slowly than those of an imperative nature. Functional languages are not suitable for general-purpose programming since it is often difficult to translate a process into a function.

The functional paradigm is seen by many as the way of the future and these drawbacks will probably be overcome.

LISP

LISP remains the principal programming language for artificial intelligence and ranks as the second oldest general-purpose language.

List processing and procedures are written quickly and revised frequently.

LISP is an acronym for '**list** **p**rocessor' and was first designed and implemented by John McCarthy and a group at the Massachusetts Institute of Technology in the late 1950s. It is one of the oldest computer languages still in widespread use.

LISP was developed out of a need predominantly in the area of artificial intelligence.

Because of its age, LISP has had many variants developed, but in the 1980s there was an attempt to standardise the language. The result of this was common LISP, and this has developed into the most popular version.

LISP is different from most other languages in a number of aspects. LISP programs run in an interactive environment and, as a result, a main program does not exist in the usual form; instead the user enters the main program as a sequence of expressions to be evaluated.

LISP is usually an interpreted language. What this means is that it is unlike other languages such as C or Pascal which are compiled languages.

A simple valid LISP expression follows:

```
(member 'x'(w x y z))
```

returns a value T (true).

The processing could be viewed as a loop, which reads what you have written, evaluates it (acts upon it) and then prints the result, before providing you with an opportunity to enter more expressions.

The original version of LISP contained only two types of objects: atoms and lists. Atoms are represented as sequences of characters. Lists are specified elements with parentheses. Lists may contain either other lists or atoms as members.

An example of a simple list could be:

```
(A B C D).
```

To assign a value to an atom you would use 'setq', thus:

```
(setq shoe_size 11).
```

In this code you have assigned the value 11 to the atom shoe_size. The interpreter would respond by assigning the atom the value 11. Thus *shoe_size* … would now elicit a response of 11.

Lists are treated as having the name of the function followed by the arguments to the function, as seen previously. Now (+*shoe_size* 3) will elicit the response 14.

LISP has always been associated with the early artificial intelligence promoters, who in the 1960s and 1970s offered artificial intelligence as a solution for all ills. That they were able to develop so little was at times blamed unfairly on LISP. Their expectations, which were too high, had nothing to do with the effectiveness and deliverability of expert-system-based applications.

One of the most important derivatives of LISP was the language LOGO. This language was invented by Seymour Papert to teach children programming. LOGO is one of the more accessible functional languages.

APL

APL is an acronym for '**a** **p**rogramming **l**anguage', which was the name given to the book in which Kenneth Iverson described his language. Operators have the same level of precedence—left to right.

APL is made up of a set of 95 characters and symbols and flow lines. The intent by Iverson was not to develop a programming language but rather to develop a notation adequate to concisely express algorithms in mathematics, which might then be easily translated into conventional programming languages.

Because of the 95 special characters and the associated symbols APL can be extremely difficult to read. But these special characters which can lead to confusion in reading APL are also seen as its strength. A single line of only a few characters can accomplish an extremely large amount of computations.

The interactive nature of APL and this conciseness means that the language is particularly attractive to a programmer who wishes to get on to the computer to test a computation, obtain the desired results and get off in a minimum of time. The language is thus less suitable for the construction of large programs that will be used repeatedly.

To enhance the aim of getting onto the computer and getting off as quickly as possible, there is no notion of main program in APL.

Within APL, expressions have no hierarchy of operations. They all have the same level of precedence, and are governed only by the laws of associativity, which rather than left to right, is right to left. For example, in an expression such as X Y + Z, with the values of X=7, Y=6 and Z=4, the expression would carry out the addition first then the multiplication, thus giving an answer of 70.

APL is unique in the sense that it is interactive with the programmer creating and executing the main program line by line during a period of time in front of the terminal.

APL has a dedicated group of followers, but it has never had the large following of other languages. As well, it has been overshadowed by the offerings from Bell Laboratories.

Exercise 9.5

1 Fill in the spaces with the most appropriate word from the following list:
LISP, recursion, modularisation, constant, functions, efficiency, loops
   a Each expression in a functional program is a ________.
   b A functional language cannot have ________ because of its inability to have variables.
   c ________ is the calling of a function or a procedure by itself.
   d ________ can be either simple assignment statements or joined together to make more complex ones.
   e Functional programming allows for greatly improved ________.
   f The two main drawbacks for functional programming are ________ and suitability.
   g ________ is the acronym for list processor.

2 Explain the difference between a compiled program and an interpreted program. In your explanation include why an interpreted program is slower.

3 Explain, in your own words, the meaning of the term 'artificial intelligence'.

4 Explain how a user enters a program in LISP.

EXTENSION EXERCISES

5 Investigate the roles of John McCarthy and Kenneth Iverson in the development of functional programming languages.

6 List the languages that have come from the Bell Laboratories. Briefly describe the purpose of each of these languages.

7 Investigate the language APL, researching the meanings of some of the 95 special characters.

Review exercises

1 Choose the most appropriate word or phrase to complete each of the following sentences.
   a The programming paradigm known as the __________ paradigm employs variables and assignments.
   b A cooking recipe is a form of __________ programming.
   c In the Von Neumann model, __________ are used to represent memory locations.
   d When an object receives a message, a __________ processes that message.
   e Pascal is an example of a structured __________ language.
   f The data stored within an object is able to be manipulated by its __________.
   g A __________ language states a number of facts, allowing the programmer to use those facts.
   h Early __________s were intended to create computer programs that were independent of the processor being used.

2 The language BASIC was chosen for early personal computers. What features of the BASIC language made it suitable for this purpose.

3 Determine the most appropriate generation language to use for each of the following applications and give reasons for your choice:
   a A stock program for a shop.
   b The ROM in a home computer.
   c A program to choose Lotto numbers.
   d A program to diagnose a patient's medical condition.
   e The electronic controller for an airconditioner.
   f The compiler for a programming language.

4 Each generation of computer language has its advantages and disadvantages. Choose two of the generations and, using an example from each, compare their advantages and disadvantages.

5 Outline the features of each of the different generations of language. How has the development of computer technology influenced the development of each generation of language?

6 Explain why it is difficult to implement a fifth-generation language on a Von Neumann style computer. Which programming paradigm is most suited for this type of computer? Give reasons for your answer.

7 Describe an application for which event-driven programming methods are most appropriate. Justify your answer.

8 Research and find an application in which APL is used and describe how this application was developed.

9 Find an application in which an expert system is used. Describe the way in which it is being used and the way in which it was developed.

10 Each of the five generations of computers has a definite difference. Draw up a table showing the main aspects of each generation and the developments from the previous generation.

11 Search the Internet and journals and write a short report showing the developments that have occurred in fifth-generation languages and future directions.

12 What are the differences between logic and object-oriented paradigms. Draw up a table showing these differences.

EXTENSION EXERCISES

13 Choose a third-generation language and investigate the different control structures for that language. What similarities can you find between these structures and those of a first-generation language?

14 Research the changes in hardware between the first and second generation of computers and outline the reasons for the resulting changes in software design.

15 Investigate the life of one of the following people and present the results of your investigation in a PowerPoint presentation: Niklaus Wirth, John Backus or Seymour Papert. In your answer outline the languages they were associated with and the effects they have had on the programming community.

16 Research emerging technologies in language development. Report back to your class explaining what you have discovered.

17 What do you believe constitutes a good programming language? Give reasons for your answer.

Chapter summary

- Computer languages can be categorised into five generations.
- First-generation languages represent instructions and data in binary form.
- First- and second-generation languages are processor dependent, resulting in programs that are written for a particular processor.
- First-generation programs are written as numbers, with binary, octal and hexadecimal notations being the most common. They are immediately executable by the processor, having no need for translation.
- Machine code is difficult to use on processors with a large word size.
- Machine instructions can be placed in five different categories: data transfer instructions, data processing instructions, test instructions, input/output instructions and control instructions.
- Second-generation languages, called symbolic assembly languages, use code words called mnemonics to represent instructions and data.
- Symbolic assembly language programming uses mnemonics to represent processor instructions; the assembler, however, calculates the addresses of data locations.
- Assemblers often contain aids such as debuggers, which assist the programmer to produce a program that functions properly.
- Assembly language programming is widely used to create interfaces to various devices such as compact disc players.
- Assembly language is not easy for the average computer user to learn.
- Third-generation and later languages are based on natural languages or symbolic languages and are used to write programs in terms of the problem being solved.
- Third-generation and later languages allow algorithms to be coded in a form that is independent of the machine being used.
- Third-generation languages are the first of the so-called high-level languages and are not bound by the processor used to execute the program.
- Various third-generation languages were influenced by the types of application they were intended to produce.
- Third-generation languages were intended to be portable between computers. However, slight differences in the languages created to take advantage of features available on a particular computer made the programs less portable.
- Fourth-generation languages may contain some of the same structures as third-generation languages, plus other mechanisms.
- Fourth-generation languages allow the programmer to concentrate more on what is to be done than on how it is to be done.
- Fifth-generation languages generally involve artificial intelligence techniques, allowing the programmer to code knowledge which the computer can draw inferences from.
- Fifth-generation languages are based on an approach known as logic programming.
- In procedural languages the program is written as a sequential set of instructions. Most first-, second- and third-generation languages are procedural.
- Procedural languages contain three main types of construction: sequences, branching and repetition.
- Declarative languages state a number of facts, which the programmer can use to state queries and pose problems. A declarative language is one form of a non-procedural language.

Chapter summary

- The common view of programming is that it consists of operations being carried out on data. Object-oriented languages do not follow this model, as they consist of objects and messages.
- An object is a block of information together with a description of the way in which it can be manipulated.
- A class is a number of objects that share common properties.
- Subprograms that define the operations on objects of a class are called methods.
- Inheritance is the ability of the client to have all of the characteristics of the parent and to gain new ones.
- Polymorphism is the ability to add a subclass used for the effective maintenance of a subprogram.
- Messages are the subprograms that define the operations on objects of a class.
- Encapsulation is the process whereby the combination of data and procedures within an object are made invisible to other objects.
- Sequential programming is used to create a set of steps which it is hoped will solve the problem.
- Event-driven programming caters for different reactions to external happenings detected by the system.
- The imperative programming paradigm is a form of sequential programming in which variables are used to model memory locations and assignments that model data transfer processes.
- Event-driven programming is suited to real-time computer applications such as control systems and simulations.
- Logic programming is very closely associated with artificial intelligence.
- Backward chaining is a problem-solving procedure working from a statement backwards.
- An inference engine provides the reasoning ability on which an expert system acts.
- A goal is a matching of rules and facts.
- The heuristic approach enables the generation of many different solutions.
- A knowledge base is a database of facts and is dynamic.

chapter 10

The software developer's view of the hardware

Outcomes

A student:

- explains the interrelationship between hardware and software (H 1.1)
- describes how the major components of a computer system store and manipulate data (H 1.3)
- identifies and evaluates legal, social and ethical issues in a number of contexts (H 3.1)
- constructs software solutions that address legal, social and ethical issues (H 3.2)
- identifies needs to which software solutions are appropriate (H 4.1)

Students learn about:

Representation of data within the computer

- binary numbers
- hexadecimal numbers
- character representation—ASCII
- representation of integers
  – sign and modulus
  – one's complement
  – two's complement
- representation of fractions
  – floating point and real numbers
- binary arithmetic
- addition
- subtraction using two's complement
- multiplication using shift and add
- division using shift and subtract

Electronic circuits to perform standard software operations

- logic gates—AND, OR, NOT, NAND, NOR, XOR
- truth tables
- circuit design steps
  - identify inputs and outputs
  - identify required components
  - check solution with truth table
  - evaluate circuit design
- specialty circuits including:
  - half-adder
  - full-adder
  - flip-flops as a memory store

Programming of hardware devices

- the input data stream from sensor and other devices
  - header information
  - data characters
  - trailer information
  - control characters
  - hardware specifications
  - documentation
- processing of the data stream
  - the need to recognise and strip control characters
  - counting the data characters
  - extracting the data
- generating output to an appropriate output device
  - required header information
  - required control characters
  - data
  - required trailer information
- control systems
  - responding to sensor information
  - specifying motor operations
- printer operation
  - control characters for features including page throw, font change, line spacing
- specialist devices with digital input and/or output

Students learn to:

- convert integers between binary and decimal representation
- interpret the binary representation of data
- recognise situations in which the data can be misinterpreted by the software
- perform arithmetic operations in binary
- generate truth tables for a given circuit
- describe the purpose of a circuit from its truth table
- design a circuit to solve a given problem and use a truth table to verify the design
- explain how a flip-flop can be used in the storage and shifting of a bit in memory
- build and test a circuit using integrated circuits or use a software package
- simulate the testing of a circuit for both user-designed circuits and the specialty circuits
- recognise the cyclical approach to circuit design
- modify an existing circuit to reflect changed requirements
- interpret a data stream for a which specifications are provided
- generate a data stream to specify particular operations for a hardware device for which specifications are provided
- modify a stream of data to meet changed requirements, given the hardware specifications
- cause a hardware device to respond in a specified fashion

This chapter looks closely at the interrelationship between hardware and software by examining the way the data is stored and manipulated in the major components of a computer system. It looks at the way a simple two-value system can be used to represent numbers and the ways numbers can be operated on. It shows how simple electronic components can be combined to carry out some of these operations. And it looks at the way streams of data flow between computing devices.

Representation of computer data

The purpose of computers and other information-technology devices is to store, process, transmit and receive data of different kinds. In all current digital computers this data is represented using a binary system. A binary system is used because there are many different physical phenomena that can take two values. For example:

- a switch can be on or off
- a light can be on or off
- a magnet can be magnetised in one direction or another
- a voltage can be high or low
- on a compact disc there can be a pit or land.

The two values of a binary system can be interpreted in different ways. For example, you could think of a high voltage as meaning 'true' and a low voltage as meaning 'false'. The most common way of writing the two possible values of a binary system is with *binary digits*. These can be one of two values: zero (0) or one (1). A binary digit is often referred to by the abbreviation '*bit*'.

Of course the world would be a fairly boring place if everything could take only two values. In fact you might say things would be black and white. Data that takes more than two values can be represented by groups of binary digits or bits. For example, two bits could be used to represent four colours:

```
00 white
01 red
10 green
11 black
```

A group of eight bits is called a 'byte'. A group of four bits is called a 'nybble'. Groups of binary digits can be used to represent any kind of data, including numbers, text, pictures, sound and computer programs. The next sections look in some detail at how binary digits can be used to represent numbers and text.

Decimal numbers

Numbers are normally written using a positional decimal system. It is called a decimal system because it is based on ten (probably because our ancestors had ten fingers) and uses ten symbols (0 to 9). It is called a positional system because digits in different positions have different meanings. You can tell that in the number 1023 the digit 3 stands for three units, the 2 stands for 2 tens and the 1 stands for 1 thousand by the position these digits have in the number. The zero is important because it holds the hundreds position even though there are no hundreds in this number. The columns in a decimal number stand for powers of ten.

ten^4	ten^3	ten^2	ten^1	ten^0
10 000	1000	100	10	1

In a number written in a positional system (for example 654 321), the right-most digit (1) is called the least significant digit and the left-most digit (6) is called the most significant digit.

Binary numbers

Numbers can be represented by binary digits using the same positional system as decimal numbers. One difference is that binary numbers are based on powers of two instead of powers of ten. The other difference is that decimal numbers require 10 symbols (0 to 9)

128	64	32	16	8	4	2	1	
0	0	0	0	0	1	1	1	= decimal 7
1	0	0	0	0	0	0	1	= decimal 129
0	0	0	1	0	0	1	0	= decimal 18
1	1	1	1	1	1	1	1	= decimal 255

Table 10.1 Some examples of binary numbers

whereas binary numbers require only two symbols (0 and 1). Some examples of binary numbers are shown in Table 10.1.

For decimal numbers, ten is called the *base* or *radix*. For binary numbers, the base or radix is two. To distinguish between numbers with different bases a subscript is used. For example:

$$1100_2 = 12_{10}$$

Converting binary numbers to decimal numbers

To convert a binary number to a decimal number you simply need to add the appropriate powers of two. For example to convert 100101_2 to a decimal number:

$$\begin{aligned} 100101_2 &= 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 32 + 4 + 1 \\ &= 37_{10} \end{aligned}$$

Converting decimal numbers to binary numbers

Copy Table 10.3 from the next page, fill it in and keep it by you. Eventually you should be able to mentally convert decimal numbers up to at least fifteen.

A general algorithm for converting decimal numbers to binary numbers involves repeated division by two. In the following examples, the second conversion shows the normal way of setting out the algorithm on paper.

Example 1

Convert decimal 6 to a binary number:

6 divided by 2 gives 3 with remainder 0
3 divided by 2 gives 1 with remainder 1
1 divided by 2 gives 0 with remainder 1

Reading the remainders from the bottom up gives the binary representation of 6 to be 110.

Convert decimal 173 to a binary number—the working is shown in the table.

The binary equivalent is found by reading the remainders from the bottom up. (The most significant binary digit is the remainder from the last division.)

$$173_{10} = 10101101_2$$

2	173	Remainder
2	86	1
2	43	0
2	21	1
2	10	1
2	5	0
2	2	1
2	1	0
2	0	1

Table 10.2 Algorithm for converting decimal numbers to binary

Hexadecimal numbers

Binary numbers might be fine for computers but they are awful for people. They use only two symbols and the string of digits can be very long, even for small numbers. Decimal numbers, on the other hand, are okay for people but they are not very useful for working with computers, because they do not have a very close relationship to a binary system.

Decimal	Binary			
	8	4	2	1
	2^3	2^2	2^1	2^0
0	0	0	0	0
1	0	0	0	1
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

Table 10.3 Binary numbers from 0 to 15

Decimal	Hexadecimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

Table 10.4 Comparing number systems

For these reasons, people who work with computers often use number systems based on powers of two, such as eight (octal numbers) or sixteen (hexadecimal numbers). Octal and hexadecimal numbers have a very close relationship with binary numbers, but it is important to realise that computers don't 'use' octal or hexadecimal numbers—everything at the fundamental level of a computer is done with binary digits. Octal or hexadecimal numbers are just a convenient way of representing what is going on with binary numbers inside a computer.

Hexadecimal numbers need sixteen symbols and are based on powers of sixteen. In hexadecimal numbers 10 means sixteen. The convention is to use the letters A to F for the numbers ten to fifteen.

Converting hexadecimal numbers to decimal

To convert hexadecimal numbers to decimal numbers, add up the appropriate powers of sixteen. For example:

$$\begin{aligned} 1A03_{16} &= 1 \times 16^3 + 10 \times 16^2 + 0 \times 16^1 + 3 \times 16^0 \\ &= 1 \times 4096 + 10 \times 256 + 0 + 3 \times 1 \\ &= 4096 + 2560 + 3 \\ &= 6659_{10} \end{aligned}$$

Converting decimal numbers to hexadecimal

Practise converting small numbers (up to 16 at least) mentally. For numbers greater than 16, you can use the same algorithm given above to convert to binary numbers as it works for any base. For example: to convert 223_{10} to a hexadecimal.

16	223	Remainder
16	13	15 = F
16	0	13 = D

Answer: DF

Converting hexadecimal to binary (and vice versa)

This process shows the simple relationship between numbers which have bases that are powers of two.

To convert hexadecimal numbers to binary numbers, write each hexadecimal digit as four binary digits:

$2A01_{16} = 0010\ 1010\ 0000\ 0001_2$

To convert a binary to a hexadecimal, write the binary number in groups of four bits and convert each group of four to a hexadecimal digit (0 to F):

```
1101011  = 0110 1011
         = 6    B     as hexadecimal
```

So if two bytes of computer memory contain these 16 bits (0011 0011 0011 0000), how do we interpret them? As seen in this section, the 16 bits could be interpreted as:

```
     0011 0011 0011 0000 – a binary number
                    3330 – a hexadecimal number
or                 13104 – decimal
```

Of course these three forms are just three ways of writing the same number, and those 16 bits could be interpreted in many more ways. They could, for example, be interpreted as four *binary coded decimal (BCD)* digits, where each decimal digit is encoded as 4 bits. When interpreted in this way, those 16 bits would stand for the **decimal** number 3330. As shown in the next section, these 16 bits could also be interpreted as two ASCII digits.

Exercise 10.1

1 **a** A binary digit can take ________ possible values.
b A decimal digit can take ________ possible values.
c A group of eight bits is called a/an ________.
d A group of four bits is called a/an ________.
e In the decimal number 27456 the most significant digit is ________.
f In the decimal number 123456789 the digit that stands for the fourth power of ten is ________.
g The radix of binary numbers is ________.
h In the binary number 10101 the middle digit stands for ________.
i The third power of two as a binary number is ________.
j What number is the hexadecimal system based on?

2 An odometer-style display has three decimal digits. The right-most digit represents one unit.
a What is the largest number that can be shown?
b How many different number values can be shown?
c The counter has two buttons labelled + and –. Pressing the + button adds one to the display and pressing the – subtracts one from the display. If the display is currently showing 000, what will be shown after the – button is pressed once?
d A calculator user is in the process of entering a decimal number. The display currently shows the number 705. What will the display show after the user presses the 4 key?

e A calculator user is in the process of entering a decimal number. The display currently shows a certain number; call it *x*. What will the display show after the user presses the 7 key?

3 **a** A traffic light has three lights (red, amber, green) which in New South Wales are normally used to give three messages (stop, stop if safe, and go). How many possible messages could be given if all combinations of the three lights being off and on are used?

b What is the highest possible number that can be represented with three bits? How many different numbers can be represented with three bits?

c A three-bit binary number is used to represent a colour with the following scheme. The left-hand bit stands for red (1) or no red (0). The middle bit stands for green and the right bit stands for blue. So 101 means red, no green, blue—this mixture gives magenta. How many different colour combinations are there and what are the colours?

4 Convert the following binary numbers to decimal numbers:

a 101 **b** 110 **c** 111 **d** 100 **e** 1000
f 10000 **g** 1111 **h** 101100 **i** 1010 1010 **j** 1111 1111

5 Convert these decimal numbers to binary numbers.

a 3 **b** 4 **c** 11 **d** 73 **e** 193
f 211 **g** 255 **h** 924 **i** 1023

6 Convert the following decimal numbers to hexadecimal.

a 5 **b** 7 **c** 8 **d** 15 **e** 10
f 23 **g** 75 **h** 1000 **i** 5678

7 Convert the following hexadecimal numbers to decimal numbers.

a F **b** 13 **c** 1B **d** 4C **e** 7F
f FF **g** 100 **h** A01C **i** FFFF

8 Convert the following binary numbers to hexadecimal.

a 11 **b** 101 **c** 1010 **d** 10000 **e** 111111
f 101101 **g** 11001100 **h** 1110000 **i** 11101011 **j** 1100110101101101101

9 **a** Draw up a table of values used in the following algorithm and check it with test data of decimalNumber = 23. Try other values for decimalNumber.

b Write an algorithm for the subprogram `Print Array of Binary Digits Bit[ ]`.

```
An algorithm to convert a decimal number to a binary number
and print out the result.
BEGIN MAIN PROGRAM
    Input decimalNumber// the number to be converted
    Convert DecimalNumber To Binary
    Print decimalNumber 'as a binary number is'
    Print Array of Binary Digits Bit[ ]
END MAIN PROGRAM

BEGIN SUBPROGRAM Convert DecimalNumber To Binary
    // Converts a decimal number to an array of binary digits.
INPUTS
    decimalNumber   //a decimal number to convert
OUTPUTS
    Bit[ ]       //an array of binary digits
    Index        //the number of digits in the array
INITIALISATION
    quotient <- decimalNumber
    Index <- 0
END INITIALISATION
    REPEAT
       Bit[Index] <- remainder of quotient ÷ 2
       quotient <- integer value of quotient ÷ 2
```

```
        Index <- Index + 1
    UNTIL quotient = 0
    Output Bit[ ] and Index
END SUBPROGRAM Convert DecimalNumber To Binary
```

Note 1: Integer value means whole number part; that is, the integer value of 3.5 is 3.

Note 2: If you want to implement this algorithm on a computer, some older values of BASIC do not have a built-in way of working out remainders. Here is a way of doing it:

```
100 REM Calculate the remainder of QUOTIENT divided by 2
110 X = INT (QUOTIENT /2)
120 REMAINDER = QUOTIENT – X*2
```

Data representation—coding methods

Representing text characters

A very common kind of data is text—data written out using the normal letters of the alphabet, numerals and punctuation marks. Text is sometimes called alphanumeric data. To store or transmit text using computing technology, there needs to be a way of coding each alphanumeric character as a pattern of binary digits (bits). There are several different standard ways of representing text data, but by far the most common is the code commonly referred to as ASCII.

A familiar example of a code for text data is Morse code. A few examples from the Morse code are:

A	· –	E	·
I	· ·	H	· · · ·
T	–	M	– –

SOS · · · – – – · · ·

V for victory · · · – (or as Beethoven said, ‘Di Di Di Dah’.)

It can be seen that the Morse code is a variable length code; that is, the codes for the different characters are not all the same length. The code was designed for operators tapping it out on a key, so codes for the most common characters are short. Morse code relies on the operator leaving pauses between characters and words. For computer communications it is more convenient to have a fixed-length code than to save time by making the common characters shorter.

ASCII code

The code for text that is used by all personal computers (and most other applications) is commonly called the ASCII (pronounced ‘ass-key’) code. ASCII stands for American Standard Code for Information Interchange. As the name suggests, the code was first used in the United States. It has since been adopted as an international standard and an Australian standard.

The code set out in the appendix is the Australian Standard Coded Character Set (AS 1776–1980). It is identical to the code specified by the International Standards Organisation, ISO 646–1973, and the Alphabet V5 defined by the International Telegraph and Telephone Consultative Committee (CCITT). It is very similar to the ISO 8859 Latin 1 encoding now commonly used, especially on the Internet. Despite all this, most people still call it ASCII code.

The ASCII code table in the appendix shows how to translate a character into a pattern of seven bits. For example, the character ‘A’, which is on the second page would be represented by the bit pattern 100 0001. The table also shows the corresponding hexadecimal (base 16) and decimal numbers (base 10) for each character. For example, the character 'k', which is on the third page, would be represented by the following:

$110\ 1011_2 = 6B_{16} = 107_{10}$

There are some things to note about the design of the code. First, there is a correspondence between alphabetical and numerical order. A common task for a computer is sorting. This correspondence means that sorting into alphabetical order can be achieved with much the same algorithms that are used for sorting numbers. (One code being less than the other means that the corresponding letter comes earlier in the alphabet.)

Second, there is a correspondence between upper-case and lower-case letters. The code for 'A' is 32 less than the code for 'a' and the same relationship holds for all the other letters in the alphabet. If you look at the binary codes for the letters, you will see that the codes for upper-case and lower-case letters differ by one bit and it is always the same bit.

There are two kinds of character shown in the table: printing (or graphic) characters and non-printing (or control) characters. Control characters have some function to do with the control of equipment or the control of transmission of data; they do not result in a visible character. Graphic characters correspond to the visible characters—the alphabetic characters, the numerals and various punctuation marks.

The control characters are in the first thirty two positions and the DEL (delete) character in the last position in the table. All of the other positions in the table hold graphics characters. The SP (space) character is an interesting one; it results in no visible marks and has an effect similar to the format effectors (it moves along one character position). However, it has a very important part to play in providing information and so is usually thought of as a graphic character rather than a control character. The ASCII control characters will be discussed further later in the chapter.

Seven or eight bits

It must be stressed that ASCII is a seven-bit code. This means that there are 128 different codes (decimal 0 to 127) that can be used to stand for characters. The code is most often used with one code to a byte (that is, one seven-bit code to an eight-bit byte). This means that there is usually one spare bit in each byte. If all eight bits in each byte were used, there would be 256 different codes that could be used for characters. The fonts used on modern computers usually use these extra 128 codes to stand for extra characters, but unfortunately Microsoft Windows and Apple Mac-OS do so in different ways. The World Wide Web specifies the ISO-8859-1 'Latin 1' standard which is again somewhat different from both Mac and Windows. The World Wide Web also uses 'Unicode', a 16-bit (double byte) standard which allows for characters from all of the main languages of the world. You can read about these character encodings at www.w3.org and www.unicode.org.

Representing numbers using ASCII

At the end of the section on binary and hexadecimal numbers it was seen that the 16-bit string of bits 0011 0011 0011 0000 could be interpreted as the number $3033_{16} = 13104_{10}$. The string could also be interpreted as a string of ASCII codes, which means it could stand for the decimal number 30 or even the hexadecimal number 30 ($= 48_{10}$).

Of course all numbers that are entered into a computer by people (through devices such as keyboards) and all numbers output from a computer for people to read (on the screen or on paper) are in the form of text. So numbers can be represented in the memory of a computer in two fundamentally different ways: as the text representation of the numeral (in an encoding such as ASCII) or as some representation of the value of the number (perhaps as a binary number). The next section looks at some variations on binary numbers to represent the values of integers and fractions.

Exercise 10.2

1 a ASCII stands for ________.
  b ASCII is a/an ________ bit code.
  c What is the difference between the ASCII code for an upper-case letter and its lower-case equivalent?
  d How many different character codes (counting control codes) are possible in ASCII?
  e Which sequence of bits begins the binary ASCII code for all decimal digits?
  f What is the first digit of the hexadecimal ASCII representation of all of the decimal digits.

2 Write each of the following decimal numbers in:

i binary	ii hexadecimal	iii decimal ASCII	iv hex ASCII
a 27	b 127	c 5	

3 Show how the following two lines of text would be encoded in ASCII in memory or a disk file. There needs to be a line fee (LF) and carriage return (CR) at the end of each line.

It's great
to be alive!

4 Does your computer system use an extended version of ASCII where all eight bits in each byte can be used for different characters? Here is a simple BASIC program you can use to find out. These programs start printing at ASCII 32 (the space) because printing the control characters can mess up a printout. (Why?)

```
BASIC
10   FOR code = 32 to 255
20   PRINT code, CHR$(code)
30   NEXT code
```

5 This algorithm takes an ASCII code and determines whether it represents a lowercase letter.

```
INPUT
  inputCode   the input ASCII code
OUTPUT
  result      is true if inputCode stands for a lowercase
              letter, false otherwise
BEGIN PROGRAM
  IF inputCode >= the ASCII code for 'a' AND
     inputCode <= the ASCII code for 'z' THEN
  result is true
  ELSE
  result is false
  END IF
END PROGRAM
```

Design an algorithm that takes as input an ASCII code and outputs the same code, unless the code was for a lower-case letter, in which case it is converted to the corresponding upper-case letter.

6 Design an algorithm that takes as input an ASCII code and outputs 'true' if the code stands for a hexadecimal digit and 'false' otherwise.

Integer binary arithmetic

Addition

The algorithm for adding binary numbers together is precisely analogous to the algorithm you use every day for adding decimal numbers. For decimal numbers a brief version of the algorithm would go something like this:

For each column of digits, starting at the right-most column:
Add the digits in the column.
If the sum is greater than 9 then carry the excess into the next column.

Example

$$\begin{array}{lr} & 9\;0\;3\;5\;+ \\ & 1\;9\;8\;3 \\ \text{Carry} & 1\;1\;0\;- \\ \hline \text{Sum} & 1\;1\;0\;1\;8 \end{array}$$

Of course this algorithm relies on you having a look-up table of the result of adding all pairs of digits (which you have learnt off by heart or stored in the ROM part of your brain). So for adding binary numbers you will have to learn that 0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 10 and 1 + 1 + 1 = 11. (That shouldn't be hard, should it?) The algorithm for adding binary numbers is exactly the same as the one above except for one change:

For each column of digits, starting at the right-most column:
Add the digits in the column.
If the sum is greater than 1 then carry the excess into the next column.

Example

$$\begin{array}{lr} & 1\;0\;0\;1\;1\;0\;1\;0\;+ \\ & 0\;1\;0\;1\;1\;1\;1\;1 \\ \text{Carry} & 0\;0\;1\;1\;1\;1\;0\;- \\ \hline \text{Sum} & 1\;1\;1\;1\;1\;0\;0\;1 \end{array}$$

It is possible to get a sum that carries over past the left-most digit you are using. That is, if you are using eight bits to store each number it is possible for the sum of two numbers to be nine bits long. This condition is called 'overflow'.

Subtraction

So far, the ways of representing numbers using binary digits have been restricted to positive integers. The following methods are used to represent both positive and negative integers.

Sign and modulus

The most obvious solution to the problem of representing negative numbers (such as –3) is to use one bit (the sign bit) to stand for the negative sign. For example, using one byte, –3 becomes 1000 0011.

In this method the left-most bit is used for the sign (1 for –, 0 for +) and the modulus (or size of the number) is represented as a normal binary number using the remaining (seven) bits. Note that both 1000 0000 and 0000 0000 would be interpreted as zero. A nice side effect of representing numbers this way is that you can check whether a number is positive or negative by checking one bit (the sign bit), as long as you are careful about zero.

Using one byte, the biggest positive integer that can be represented this way is 0111 1111, or 127_{10}. The smallest negative integer is 1111 1111, or -127_{10}.

Excess form

This method uses a starting point to stand for zero. The starting point is the number with the most significant bit equal to 0 and all other bits equal to 1. For eight-bit numbers the starting point will be $0111\ 1111_2 = 127_{10}$.

To convert a number from its coded form, we calculate the amount by which the coded form exceeds the starting point. For example:

1000 0010 stands for 130 – 127 = +3
0111 1110 stands for 126 – 127 = –1

Note that positive numbers have a 1 in the most significant bit, and negative numbers a zero. The biggest number eight bits is 255 – 127 = 128; the smallest is 0 – 127 = –127.

Ones and twos complements

Another (more useful) way of representing negative numbers uses complements. To understand complements, it is helpful to work briefly in decimal numbers and to think of an example of a register such as the odometer in a car.

If the odometer is showing 000000 and it was caused to go backwards by one, it is not hard to imagine that it will then show 999999. Therefore 999999 can be used to represent –1, 999998 to represent –2, 999997 to represent –3 and so on. The number 999999 is called the tens complement of 1.

To calculate the tens complement of a number:

- Calculate the nines complement of the number and then add one. If there is a carry beyond the number of digits in the register, ignore it.

To calculate the nines complement of a number:

- Subtract each digit from 9.

Example

What are the nines and tens complements of 000237?

```
                      9 9 9 9 9 9
                      0 0 0 2 3 7
                      -----------
Nines complement      9 9 9 7 6 2 +
                      0 0 0 0 0 1
                      -----------
Tens complement       9 9 9 7 6 3
```

So –237 can be represented as 999762 using the nines complement, or 999763 using the tens complement.

If you take the negative of the negative of 237 you get back to 237. So if you take the tens complement of the tens complement of 237 you should get back to 237. Do you?

Use the same scheme as with the binary number system (imagine an odometer with just 1s and 0s) except now use the twos and ones complements.

To calculate the twos complement of a binary number:

- Calculate the ones complement of the number and add one. If there is a carry beyond the number of digits in the register, ignore it.

To calculate the ones complement of a binary number:

- Subtract each digit from 1 (shortcut—swap ones and zeros).

Example

Show how to represent the negative of the binary number 0010 1101 using ones complement and twos complement.

```
                      1 1 1 1 1 1 1 1
                      0 0 1 0 1 1 0 1
                      ---------------
Ones complement       1 1 0 1 0 0 1 0 +
                      0 0 0 0 0 0 0 1
                      ---------------
Twos complement       1 1 0 1 0 0 1 1
```

What are the ones and twos complements of 0000 0000?

```
                      1 1 1 1 1 1 1 1
                      0 0 0 0 0 0 0 0
                      ---------------
Ones complement       1 1 1 1 1 1 1 1 +
                      0 0 0 0 0 0 0 1
                    -----------------
                    1 0 0 0 0 0 0 0 0   ignore carry
Twos complement       0 0 0 0 0 0 0 0
```

Just as with the sign and modulus method, you can check whether a number is positive or negative just by checking the left-most bit. It will be 1 for a negative number and 0 for a positive number. (Strictly speaking, you should say that if the left-most bit is 0, the number is non-negative rather than positive, because the number could be zero.)

There are two good reasons for using complements (usually twos complement). First, they are easy to calculate—most of the process just involves swapping zeroes and ones. Second, they have a simple algorithm for doing subtractions.

To subtract one binary number from another:

- Take the twos complement of the number being subtracted and add it to the first number. If there is an overflow, ignore it.

Example

Calculate 0010 1010 – 0001 0011 (that is, 42 – 19 = 23).

```
                            1 1 1 1 1 1 1 1
The second number           0 0 0 1 0 0 1 1
                           ----------------
Ones complement             1 1 1 0 1 1 0 0 +
                            0 0 0 0 0 0 0 1
                           ----------------
Twos complement             1 1 1 0 1 1 0 1 +
The first number            0 0 1 0 1 0 1 0
                          -----------------
                          1 0 0 0 1 0 1 1 1  ignore overflow
The difference              0 0 0 1 0 1 1 1
```

Note that this method of subtraction assumes that all numbers are represented in the twos complement form; that is, any number (in the question or the answer) with one in the left-most column is negative. If you do the above subtraction the other way round (0001 0011 – 0010 1010), you should get a negative answer. Check it.

Multiplication

As with addition, the algorithm for multiplying binary numbers is exactly the same as that for decimal numbers. Before looking at a general algorithm, recall the very simple trick for multiplying decimal numbers by ten. This is normally stated as, 'To multiply by ten, add a zero'. It would be a little more precise to say, 'To multiply by ten, shift each digit one place to the left'. For example:

$$1234 \times 10 = 12\,340$$

Similarly, in the binary number system you can multiply by two (the base) by shifting each digit one place to the left. For example:

$$110110 \times 10 = 1101100$$

The general algorithm for multiplying decimal numbers relies on knowing the multiplication tables up to 9. Multiplying binary numbers is much easier because you only have to know multiplication tables for 0 and 1!

Example

Multiply 1011001 by 101.

```
      1 0 1 1 0 0 1 ×
      0 0 0 0 1 0 1
      -------------
      1 0 1 1 0 0 1
    0 0 0 0 0 0 0 0
  1 0 1 1 0 0 1 0 0
  -----------------
  1 1 0 1 1 1 1 0 1
```

Microprocessors have instructions for shifting binary numbers to the left and the right built into them. You can write an algorithm for multiplying numbers just using shifts and additions.

```
   Algorithm to multiply two numbers (the Multiplicand and the
Multiplier) with the result in Product.
   INITIALISATION
     Product <- 0
   END INITIALISATION
   BEGIN PROGRAM
     WHILE the Multiplier is not 0
        IF the least significant digit of the Multiplier is 1 THEN
         Add the Multiplicand to the Product
        END IF
        Shift the Multiplier one place to the right
        Shift the Multiplicand one place to the left
     END WHILE
   END PROGRAM
```

Division

Remember the quick way of dividing decimal numbers by ten by lopping off one digit? Similarly, with binary integers you can quickly divide by two by shifting all the digits one place to the right.

The algorithm you were taught to perform long division with base ten numbers uses shift and subtract operations and works just as well with binary numbers, as illustrated in the following example.

Example

Divide 110110_2 by 101_2.

```
                                  1 0 1 0
                              ___________
                          1 0 1 ) 1 1 0 1 1 0
                                    1 0 1
                                    -----
Subtract                              1 1 1 0
Shift                                 1 0 1
                                      -----
Subtract                                1 0 0
Shift                                   1 0 1
Subtraction will give negative
No further shifts are possible
```

Answer: $110110_2 \div 101_2 = 1010_2$ remainder 100_2

(This pencil and paper version of the division algorithm is all students need to know at this stage. A more complete version of the algorithm in pseudocode is given in G. Sharkey and E. Chopping, *Heinemann Senior Computing Studies 2/3 Unit Common HSC Course*, Heinemann 1995.)

Fractions and fixed point, floating point and real numbers.

You already know that when we write a number such as 1.234 as a *decimal* fraction (with a *decimal* point), we mean one unit plus 2 tenths plus 3 hundredths plus 4 thousandths (or we can read it as 1234 thousandths). We can apply the same principle to binary numbers, except, of course, we now use a *binary* point. So we can write:

$$1.10101 = 1 + \frac{1}{2} + \frac{0}{4} + \frac{1}{8} + \frac{0}{16} + \frac{1}{32} = \frac{53}{32}$$

The simplest way we can use a fixed number of bits (say 8, 16 or 32) to represent a binary fraction is to state that a certain number of bits are used for the fractional part (that is, a certain number of bits come after the binary point). Such a scheme is called a **fixed point** representation. For example, with 16 bits we might say that the first 8 bits are the whole number part and the second 8 bits are the fractional part. The disadvantage of the fixed point scheme is that we cannot represent very large and very small numbers.

There is another scheme for representing fractional numbers with a fixed number of digits, which you probably see every day on your pocket calculator. This scheme uses exponential notation and represents each number with two parts: the mantissa and the exponent. For example:

Mantissa	Exponent		
456.12783	09	$= 456.12783 \times 10^9$	$= 456127830000$
3.45678	–04	$= 3.45678 \times 10^{-4}$	$= 0.000345678$

This notation is called **floating point**, because there are many positions that the decimal point can occupy, depending on the exponent. There is a special form of this notation, called **scientific** or **standard notation**, where the mantissa is restricted to being greater than or equal to 1 and less than 10. (That is, the mantissa must have exactly one non-zero digit before the decimal point.)

There are many ways we could use a fixed number of bits to represent a floating point number, but most computer systems use a version of the IEEE 754-1985 standard produced by the Institute of Electrical and Electronics Engineers (IEEE). The **single precision** version of this standard uses 32 bits as follows (with the bits numbered from 0 to 31 from left to right).

```
S  EEEEEEEE  FFFFFFFFFFFFFFFFFFFFFFF
0  1       8 9                     31
```

- S stands for sign. If S = 0 the number is positive; if S = 1 the number is negative. That is, the number is in *sign plus modulus* form.
- E stands for exponent in excess 127 form. That is, the exponent for this number is E-127. Special meanings are used for E = 255 and E = 0.
- F stands for the fractional part of the mantissa. We form the mantissa by putting 1 and a binary point before F. Because the leading 1 is just assumed to be there, it is sometimes called the *hidden bit*.

So for $0 < E < 255$ the value, V, of the number is:

$V = +\,2^{(E-127)} \times 1.F$ when $S = 0$ or

$\quad -\,2^{(E-127)} \times 1.F$ when $S = 1$ (See example below.)

0 10000001 00000000000000000000000

$= +\,2^{(129-127)} \times 1.00000000000000000000000$

$= 2^2 \times 1$

$= 4$

1 01111111 01000000000000000000000

$= -\,2^{(127-127)} \times 1.01000000000000000000000$

$= -\,2^0 \times 1.25$

$= -\,1.25$

0 01111110 10000000000000000000000

$= +\,2^{(126-127)} \times 1.10000000000000000000000$

$= +\,2^{-1} \times 1.5$

$= +\,0.75$

Noting that 1.11111111111111111111111 is approximately 2 (think about decimal 9.999999999999 …), the biggest possible positive number is:

= 0 11111110 11111111111111111111111
$= + 2^{(254 - 127)} \times 1.11111111111111111111111$
$\approx + 2^{127} \times 2$
$\approx + 3.40282 \times 10^{38}$

The smallest positive number is:

= 0 00000001 00000000000000000000000
$= + 2^{(1 - 127)} \times 1.00000000000000000000000000$
$\approx 1.17549 \times 10^{-38}$

You should note that so far there is no way of representing the number zero.
It is now time to look at the special values of E, 0 and 255. (The following is only a partial explanation.)

- If E = 255, the value is 'infinity', or 'NaN' which stands for 'Not a Number'. Arithmetic systems should give this as an answer when the user attempts to do an illegal calculation such as 0/0 or finding the square root of a negative number.
- If E = 0, F = 0 and S = 0, the value is 0.

The IEEE standard also provides for double precision numbers using 64 bits, with 1 sign bit, 11 bits for the exponent and the remaining 52 bits for the mantissa. The double precision standard follows similar rules to the single precision standard outlined above. Other precisions are also possible; the Macintosh, for example, provides for 80-bit floating point numbers.

In several computer programming languages floating point numbers are also known as **real numbers**. In fact, in a computing context the terms 'floating point numbers' and 'real numbers' usually refer to the same thing. We should keep in mind, though, that the true mathematical meaning of the term 'real number' is any number that can be plotted on the number line (including π and the square root of 2), and floating point numbers in a computer can only ever be an approximation dpending on how much precision is used. Just as in decimal numbers some fractions have infinitely repeating decimal fractions (e.g. 1/3 = 0.3333 …), so some simple fractions in binary arithmetic have repeating forms (1/5 = 0.001100110011001100 …). This means that fractions like one-fifth, one-tenth or one-hundredth cannot be represented accurately in the standard IEEE form no matter how many bits are used. Programmers must choose a representation for numbers that will minimise errors, and they must take care that any errors that do occur do not accumulate in such a way as to become significant.

Exercise 10.3

1 **a** To multiply a decimal number by ten, shift each digit to the ___________.
b To divide a decimal number by ten, shift each digit to the ___________.
c To multiply a binary number by two, shift each digit to the ___________.
d To divide a binary digit by two, shift each digit to the ___________.
e To divide a hexadecimal digit by ___________, shift each digit one place to the right.
f To multiply an octal digit by ___________, shift each digit one place to the left.
g What is the ones complement of 0101 1101?
h To multiply a binary number by 8, shift the digits ___________ places to the left.
i If the binary number 1000 1010 is in sign plus modulus form, is it positive or negative?
j If the number 1000 0011 is in sign plus modulus form, what is the modulus?

2 Perform these additions. Show all working.
   a 0101 + 0011
   b 0110 1101 + 0101 0101
   c 0101 1011 + 0010 1010
   d 0011 1011 + 0010 0010 + 0011 0001
   e 0011 1100 1010 1111 + 0001 1111 0000 1111

3 Write these negative numbers as eight-bit binary numbers in ones complement, twos complement and sign plus modulus form.
   a -3_{10} b -99_{10} c -1010_2

4 Perform these subtractions using the twos complement method. Show all working. Write answers in twos complement and sign plus modulus form.
   a 0001 0011 – 0000 1010   b 0111 0000 – 0001 1100   c 0000 0101 – 0001 1100
   d 0000 0110 – 1000 1110   e 1000 1111 – 0101 1011

   Example: 0000 0011 – 0000 0101

The second number	0000 0101	
Ones complement	1111 1010	
Add one	0000 0001	
Twos complement	1111 1011	
Add first number	0000 0011	
Difference	1111 1110	in twos complement form

   The sign bit is 1 so the number is negative, therefore you need to take the twos complement again to get the modulus.

Ones complement	0000 0001	
Add one	0000 0001	
Twos complement	0000 0010	the modulus
	1000 0010	answer in sign plus modulus form.

5 Perform these multiplications. Show all working.
   a 0011 1010 × 10   b 0001 1011 × 101   c 0010 1101 × 1101
   d 11 × 1011 0011   e 0011 1101 × 0101 1011

6 Perform these divisions by a simple shift. Show all working.
   a 0010 1110 ÷ 10   b 0001 0000 ÷ 0000 0100   c 1100 0000 ÷ 0001 0000
   Perform these divisions by following through the shift-and-subtract algorithm given previously. Show all working.
   d 0000 1100 ÷ 0000 0011   e 0010 1100 ÷ 0000 0101   f 0111 1111 ÷ 0000 0111

7 a Write -8.0_{10} as an IEEE 32-bit floating point number.
   b Write 0.125_{10} (1/8) as an IEEE 32-bit floating point number.
   c Interpret 1 00000100 10000000000000000000000 as an IEEE 32-bit floating point number.
   d Interpret 0 00000000 00001000000000000000000 as an IEEE 32-bit floating point number.

8 Examine the pseudocode algorithm for multiplication given previously. Work through the algorithm with the multiplicand equal to 1011 and the multiplier equal to 0101. Draw up a table with headings for multiplicand, multiplier and product and show the values each time the algorithm goes through the WHILE statement.

9 A programmer has a need to perform a very large number of multiplications by 5_{10} (101_2), so she decides to write a special algorithm for the task. How might this algorithm go?

10 What changes would need to be made to the pseudocode algorithm for multiplication if it is to take account of negative numbers in twos complement form?

Electronic circuits to perform standard software operations

Computing technologies use binary digits to represent data because many physical phenomena can be present in one of two states which correspond to the binary digits, zero and one. This section looks at how the binary digits can be represented by two levels of electrical voltage and how simple electronically controlled switches (made with transistors) are used to perform computing functions. Very simple circuits are studied first, followed by complex circuits packaged into integrated circuits.

Consider a simple switch such as you would find in a battery operated torch, as shown in Figure 10.1. In logical terms, the input to this circuit is the state of the switch (which can be on or off), and the output is the level or voltage from the switch which is shown by the lamp (which can be on or off).

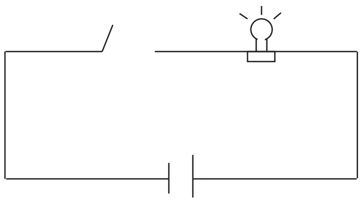


Figure 10.1 Simple circuit with switch, battery and lamp.

Transistors can be used to build simple electronically controlled switches (see Figure 10.2a). The advantage of electronically controlled switches, as we will see, is that the output from one switch can be the input to another. Often the diagram is simplified by leaving out the details of the power supply, as shown in Figure 10.2b. The details of such circuits are beyond the scope of this text, but many references are available, including a discussion in G. Sharkey and E. Chopping, *Heinemann Senior Computing Studies 2/3 Unit Common HSC Course*, Heinemann 1995.

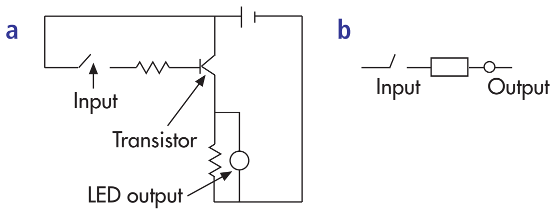


Figure 10.2 Simple electrically controlled switch.

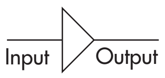


Figure 10.3 Logic diagram symbol for a buffer.

A circuit like this is called a **logic gate**—its inputs and outputs take logical values (true or false, 0 or 1). This circuit is the simplest kind of logic gate and perhaps is not a very interesting one since its output is exactly the same as its input. It is sometimes called a 'buffer' and its symbol in logic diagrams is as shown in Figure 10.3. With similar simple transistorised circuits we can make logic gates with more useful functions.

A logic gate is a circuit that takes one or more logical (or binary) values as inputs and has one or more logical (or binary) values as outputs. A logical value can have two values that can be represented as true or false, on or off, 1 or 0. In the physical circuits in this chapter the two values are a 'high' voltage (around 5 or 6 volts) for true or 1 and a 'low' voltage (around 0 volts) for false or 0.

Input	Output
0	1
1	0

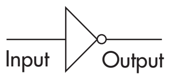


Figure 10.4 Truth table and symbol for a NOT gate.

NOT gates

Apart from a buffer, the simplest kind of logic gate is the NOT gate (also called an inverter or inverting buffer). A NOT gate has one input and one output and the output is the opposite (or complement) of the input. If the input is true then the output is NOT true and if the input is false then the output is NOT false.

We can show the relationship between the inputs and the outputs of a logic gate using a **truth table**. A truth table lists all possible inputs for the gate and shows the corresponding output. The truth table and logic symbols for a NOT gate are shown in Figure 10.4. The buffer (which does not change its input) and the NOT gate are the only possible logic gates that have only one input.

AND gates

A simple logic gate that has two inputs is the AND gate. The output of an AND gate is true if, and only if, the first input is true AND the second input is true.

To draw up a truth table for a gate that has two inputs, all of the possible combinations of values for the two inputs are listed. You will see that these combinations are the same as binary numbers from zero to three. Figure 10.5 shows the truth table and logic symbols for AND gates.

Inputs		Outputs
A	B	
0	0	0
0	1	0
1	0	0
1	1	1

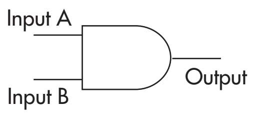


Figure 10.5 Truth table and symbol for AND gate.

OR gates

The OR gate is another with two inputs. The output of an OR gate is true if one of the two inputs is true OR the other input is true OR both inputs are true. The truth table and logic diagram symbol for an OR gate are shown in Figure 10.6.

Inputs		Outputs
A	B	
0	0	0
0	1	1
1	0	1
1	1	1

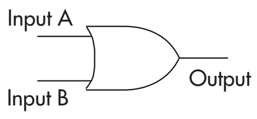


Figure 10.6 Truth table and symbol for an OR gate.

NAND, NOR and XOR gates

NAND, NOR and XOR gates all have two inputs. From the descriptions below, you should be able to draw up the truth tables for each gate.

The output of a NAND gate is the opposite of the output of an AND gate. (A NAND gate is a NOT AND gate.) The output of a NAND gate is true if, and only if, it is not true that both the inputs are true.

The output of a NOR gate is the opposite of the output of an OR gate. (It is a NOT OR gate.) The output of a NOR gate is true if, and only if, it is not true that either of the inputs is true.

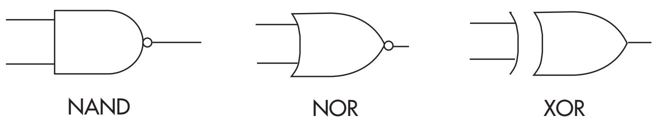


Figure 10.7 Symbols for NAND, NOR and XOR gates.

The output of an XOR gate is true if either of the inputs is true but *not* if both inputs are true. The XOR stands for 'exclusive or' and sometimes it is written as EOR.

Combining gates

It is the combination of these simple gates that will give us the functions of a typical computer. It is an amazing fact that you could build a whole computer using just combinations of NAND gates—a typical microprocessor would contain millions of them. In practice, the transistorised circuits for a number of gates are packaged into integrated circuits (ICs). If you look inside a computer you will see many ICs, mostly in the familiar form of rectangular black plastic packages with legs.

The logical functions described here could be investigated using just pencil and paper, or by actually building the circuits using transistors and ICs, or most conveniently by using a computerised simulation. A very good program for Macintosh computers is called DigSim; it is available from a number of download sites on the Internet.

The following sections look at ways of analysing and designing logic circuits for familiar functions such as adding numbers and storing data. First, the method for analysing combinations of gates is discussed.

Designing circuits

The process of designing logic circuits follows the same general process that is used for developing computer software. The steps in the process are:

- Identify the inputs and outputs. If possible, describe the desired function in logical terms (using AND, OR, NOT). Draw up a truth table for the desired function.

Example 2

Draw up the truth table for the logic diagram shown in Figure 10.8.

The first step is to list all possible combinations of the three inputs. From your study of binary numbers you know that there are $2^3 = 8$ combinations. One way to make sure you get all the combinations is to list them in numerical order as binary numbers.

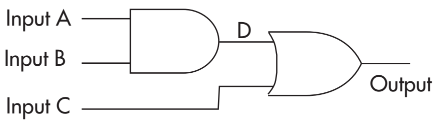


Figure 10.8 Combined gate with intermediate output–input labelled.

The second step is to label intermediate points where the output from one gate becomes the input of the next. Include a column for intermediate points in your truth table.

A	B	C	D = A AND B	Output = C OR D
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Figure 10.9 Truth table for the circuit shown in Figure 10.8.

- Identify the required components. Are there recognisable parts of the function that correspond to the logic gates we know about, either the standard gates or the integrated components (adders and flip-flops) that are discussed below. There are mathematical techniques for calculating circuit diagrams but these are beyond the scope of this course. In practice, a designer would use a computer-aided design (CAD) package to help design the circuit.
- Check the solution with a truth table. Make sure that the truth table for the proposed design matches the truth table for the required function. Again, in practice a designer would use a computerised package (such as the simulations described above) to check a design. The computerised simulation will check not only logical function but also other important factors such as timing. Modern integrated circuits are so complex that they could not be designed without the use of these CAD and simulation software packages. It is somewhat ironic that the development of modern computers could not have been achieved without the development of modern computers to use in the design process.
- Evaluate the design. Does it solve the problem? Is it economical? Can it be improved (by using fewer components for example)? The proof of the pudding is in the eating and in practice the designing engineer would make a prototype of the design to check that it works.

Example 3

Draw up a truth table for a three-input AND gate. Draw a logic diagram which shows how to make a three-input AND gate using gates with two inputs. A three-input AND gate is where the output is true if, and only if, the first input is true AND the second input is true AND the third input is true.

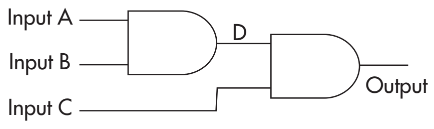


Figure 10.11 Three-input AND gate made from two 2-input AND gates.

If you label the three inputs as A, B, C, you can write:

Output = A AND B AND C
= (A AND B) AND C

The bracket form provides a clue about how to design a logic circuit with gates with two inputs. Again there are three inputs, giving $2^3 = 8$ combinations.

A	B	C	D = A AND B	Output
				= A and B and C
				= D AND C
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

Figure 10.10 Truth table for a three-input AND gate.

Adding numbers

This section looks at the way logic gates can be used to do arithmetic. Recall the algorithm you practised earlier for adding two binary numbers together. This was done by adding the numbers column by column, starting at the right-hand column, and carrying a 1 into the next column if necessary. Start by designing a logic circuit to add one column of two binary digits. You can then chain copies of your circuit to add as many columns as you need.

With two binary digits as inputs, the possible sums are:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

You can see that to allow for the case 1 + 1 your circuit will need to have two outputs—the partial sum and the carry. The truth table required is shown in Figure 10.12.

In logic terms, the sum is true if input A OR input B is true, AND NOT both A AND B are true (i.e. A XOR B). The carry is true if both A AND B are true. Without looking at Figure 10.13, try to design a logic diagram with the appropriate outputs.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 10.12 Truth table for adding two bits.

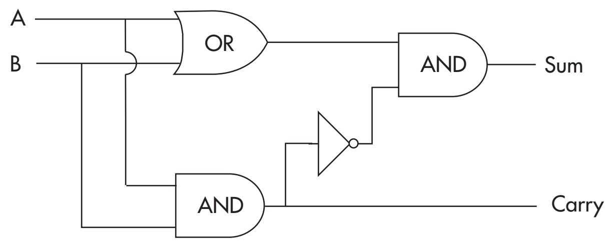


Figure 10.13 Binary half-adder.

This circuit is still not what you need to add binary numbers together, as it does not allow for the fact that there may have been a carry from a previous addition. This circuit is thus called a half-adder. A full-adder will need three inputs: the two binary digits being added and the carry.

To design a full-adder you can utilise copies of the circuit you already have for a half-adder—add the two digits together and then add the carry input. If either of the two carries is true, the final carry is true (see Figure 10.14). You should draw up a truth table for the full-adder. (It has three inputs, so the truth table will need eight lines.) You should also make sure that you understand the purpose of the full-adder sufficiently to be able to do question 7 in Exercise 10.4.

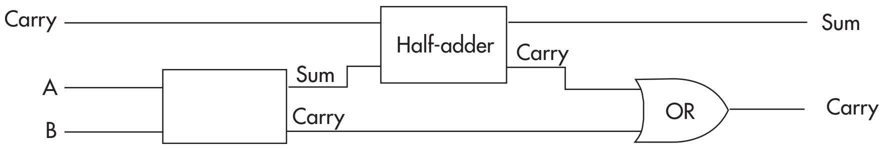


Figure 10.14 Binary full-adder.

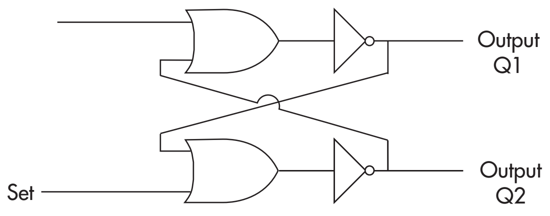
Storage of binary data

Binary data can be stored in logical elements called flip-flops. A flip-flop's basic characteristic is that it can be in only one of two possible states at any given time and that it will remain in that state until commanded to change states. Because flip-flops stay in one of two states they are called bistable devices.

The simplest kind of flip-flop is called a reset-set flip-flop or an RS flip-flop. An RS flip-flop can be made using the basic logic gates, using the circuit in Figure 10.15. Like most flip-flops it has two outputs, but one is always the complement (or opposite) of the other. Use one of the simulation programs to carefully analyse the way this circuit works. Note that the case of both inputs being 1 is not allowed.

A slightly more complicated flip-flop is the gated RS flip-flop. The circuit shown in Figure 10.16 is the same as the plain RS flip-flop with a few more gates added to the front and an extra input.

While the ENABLE input is 0, the SET and RESET inputs will have no effect on the output. When the circuit is ENABLEd (that is, the ENABLE input is 1), it will behave just like an RS flip-flop. Computer circuits are controlled by a clock pulse which synchronises the action of all individual elements of the circuit, so the ENABLE input is sometimes called the CLOCK input.



Set	Reset	Q1	Q2
0	0	(No change)	
1	0	1	0
0	1	0	1
1	1	Not allowed	

Figure 10.15 RS flip-flop with truth table.

One further addition to the circuit for a gated RS flip-flop gives the data or D-type flip-flop (see Figure 10.17). The SET input is fed through an inverter (a NOT gate) to the RESET input. The RESET input will now always be the opposite of the SET input (and vice versa). The circuit now has only one input (not counting the ENABLE). This input is called the DATA input.

If the ENABLE is on, the output from a D-type flip-flop will follow whatever the DATA input is doing. If the ENABLE is turned off, the output will stay in its current state until the ENABLE is turned on again. A D-type flip-flop is sometimes called a 'latch'.

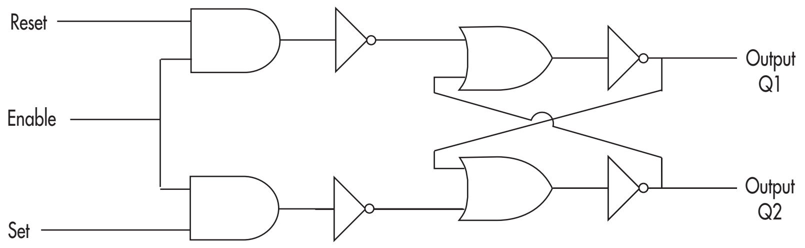


Figure 10.16 Gated RS flip-flop.

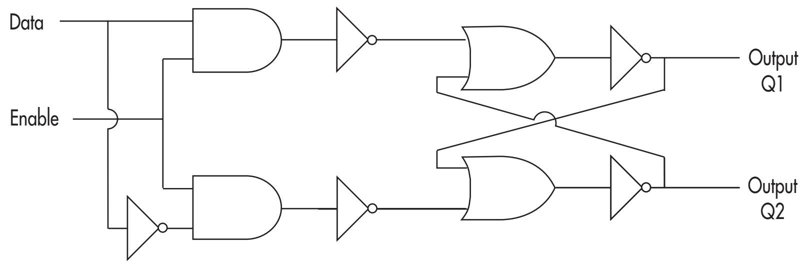


Figure 10.17 D-type flip-flop.

The D-type flip-flop can be used to store one bit of data. If more than one bit of data is to be stored, extra flip-flops must be used. To store one byte eight flip-flops would be needed. Figure 10.18 shows how eight D-type flip-flops can be combined to store one byte.

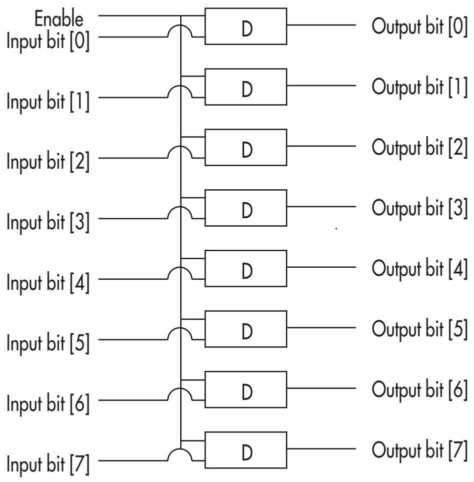


Figure 10.18 Storing one byte with eight flip-flops.

Integrated circuits

You have seen how to make simple logic gates using electronic components, and how these simple gates can be combined to perform more complex functions. You have also seen how subcircuits can be used as building blocks for other circuits (a full-adder made from half-adders, a memory for a byte made from eight D-type flip-flops, and so on).

The entire circuit for a complex function can be made into a single block. These blocks are called integrated circuits (ICs) and since the 1960s they have become ever more complex. If you look inside any electronics device (especially computers), you will see a circuit board that is mostly made up of integrated circuits. These circuits range in complexity from a package containing a few AND gates to complete microprocessors and memory chips containing the equivalent of millions of transistors. Most integrated circuits are packaged black plastic rectangles with a number of metal ‘legs’. The legs are connectors for the inputs and outputs of the circuit as well as for power.

A common type of simple integrated circuit is the 74xx series of TTL ICs. The TTL stands for transistor–transistor logic and refers to the techniques used to design the integrated circuit. If you look at integrated circuits you will see that one end is marked, usually with a notch. The pins are numbered anticlockwise starting from the mark. Figure 10.19 shows the functions of some common integrated circuits that you can buy from any electronics store for less than a dollar each.

Integrated circuits depend for their operation on transistors, which were first developed in 1947 by John Bardeen, William Shockley and Walter Brattain at Bell Laboratories. Transistors are manufactured using semiconductor elements such as silicon and germanium. Silicon is a very attractive element to use to make something useful because next to oxygen it is the most abundant element on Earth. It is the principal ingredient in sand, quartz and glass. In the 1960s engineers such as Robert Noyce at Fairchild Semiconductor and Jack Kilby at Texas Instruments developed ways of building whole electronic circuits of transistors and other components in a single block. These circuits were made up of layers of semiconductor doped with various impurities on a base of silicon (the ‘silicon chip’). Robert Noyce went on to help found Intel Corporation, the company that makes, among other things, the microprocessor chips used in most of the PCs in the world.

The development of integrated circuits has been very rapid. With improved manufacturing techniques, the complexity of the circuit that can be built onto one chip has increased many-fold. The 7408 chip with AND gates, shown in Figure 10.19, has the equivalent of about 16 transistors; the 80486 microprocessor has over 4 million transistors. Memory chips that store 4 megabits are becoming commonplace and chips storing 16 megabytes are available.

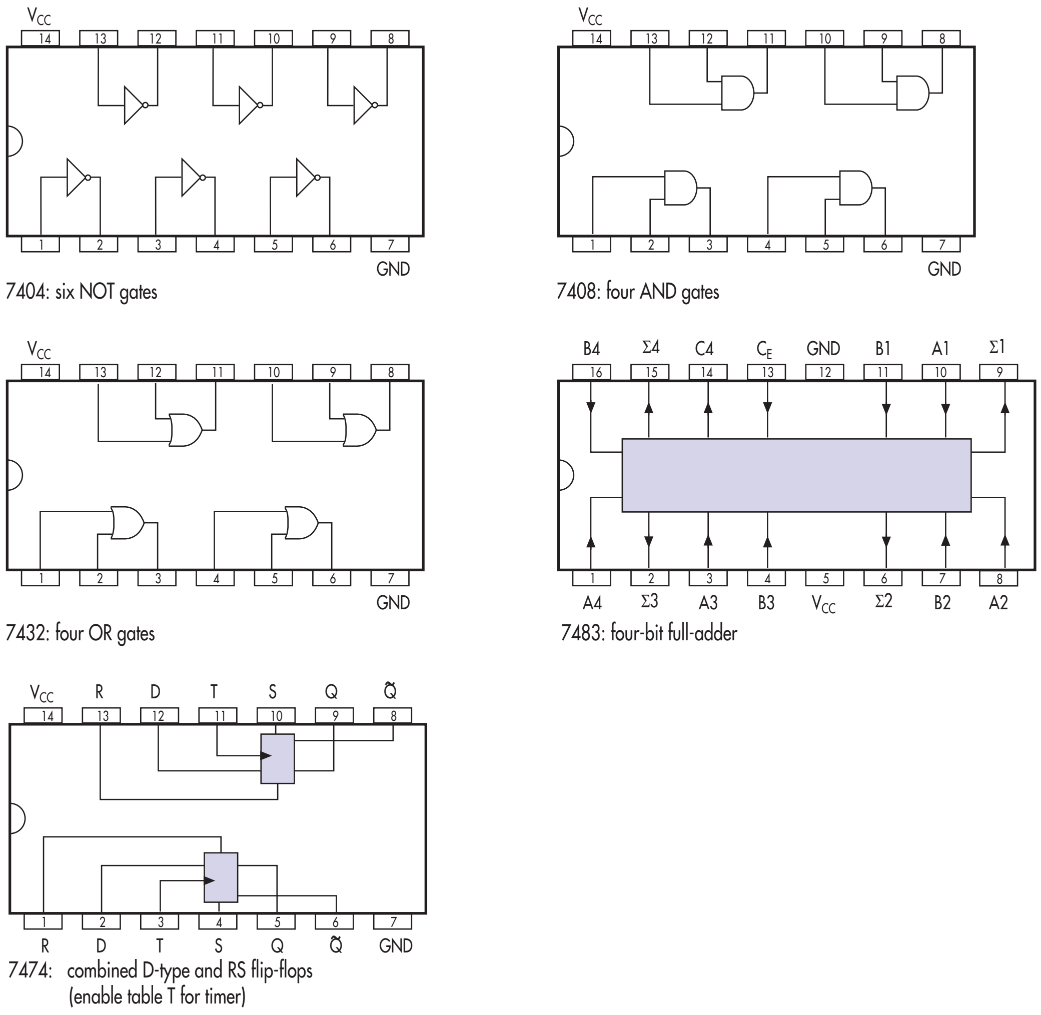


Figure 10.19 Some common integrated circuits.

Exercise 10.4

1 **a** If the input to a NOT gate is true, the output is ____________.
b If both inputs to an AND gate are true, the output is ____________.
c If both inputs to an OR gate are true, the output is ____________.
d If only one input to an AND gate is true, the output is ____________.
e A bistable device can be in one of ____________ states.
f How many half-adders are needed to make a full-adder?
g How many flip-flops are needed to store a byte of data?
h CAD stands for ____________.
i IC stands for ____________.
j Which pair of binary digits, when added together, produce a carry?

2 Give definitions in words and with truth tables for NOT gates, AND gates and OR gates.

3 Draw up truth tables for the circuits shown in Figure 10.20.

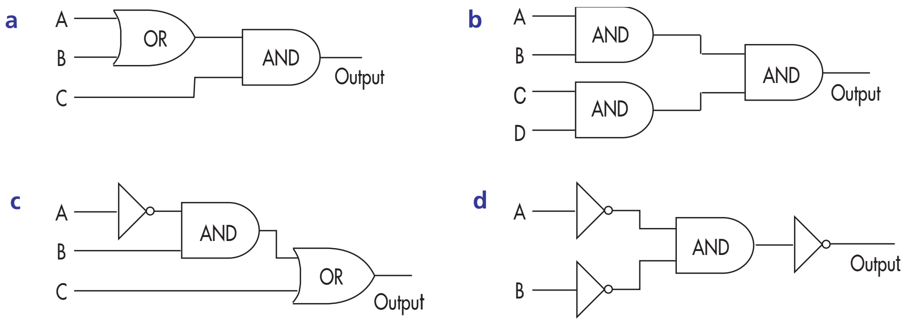


Figure 10.20

4 Draw up a truth table and a logic circuit for the following using ONLY NOT, AND and OR gates.
   - a A NAND gate.
   - b A NOR gate.
   - c An XOR gate.
   - d A quad input AND gate, which has four inputs and the output is true if, and only if, all inputs are true.

5 Draw a logic circuit for a D-type flip-flop using NANDand NOR gates.

6 Explain how a half-adder can be made from simple logic gates.

7 Draw a logic diagram showing how full-adders can be combined to add two four-bit numbers.

8 Show how the logic circuit in question 7 could be packaged into an integrated circuit. Show how two of the integrated circuits could be combined to add two eight-bit numbers.

9 A challenge
Design a logic diagram that has two inputs and four outputs according to this truth table. Such a circuit is called a decoder.

Inputs		Outputs			
A	B	C	D	E	F
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

10 More challenges
   - a Design a logic diagram with four inputs and ten outputs. Label the outputs 0 to 9. If the four bits of input are interpreted as a binary number, the appropriately labelled output should be true; all others will be false.
   - b Design a circuit that takes a four-bit binary number as input and has seven outputs that will control the appropriate parts of a seven-segment digit display (calculator-style display).

Programming of hardware devices

From your study and your experience of computer hardware you will realise that a computer system is made up of many different subsystems. The desktop computer system you use has subsystems that process input from devices such as keyboards and mice, control storage devices such as disk drives, and control output devices such as VDUs and printers. Other subsystems communicate with other computing devices through modems and networks. Most computer users do not need to worry about these subsystems, but software developers need a good understanding of the way the hardware is programmed.

Your school will probably have many hardware items that you can experiment with (and maybe develop projects for), including joysticks, science tool kits, old-style dot matrix printers, Lego control systems, data loggers, graphic calculators, plotters, CNC lathes, MIDI musical instruments and turtle robots. There are many differences in the way these devices are used, so it is hard to give specific advice about programming them. You should, of course, use all of the problem-solving skills and techniques you have learnt in this course, including making sure you have carefully read any documentation.

Some of the simplest specialised hardware input and output devices have been used to provide access to computer software for people with disabilities. Paraplegics are capable of very little physical movement, but they may be able to operate a simple switch, one operated by the breath, for example, or by a movement of the head. As well as providing access to computer software, for many these devices have provided for the first time a way of communicating with others. How would you go about designing software that will allow typing where the only input is a simple switch? See what you can find out about other kinds of special input and output devices for people with disabilities. Have a class discussion about the implications of the rights of people with disabilities to be able to use software in general. Do software designers have an obligation to design software that does not deny access to people with the need for special input devices?

Processing the input

The input to a system may come from some other processing system (for example the main input to a printer is generated by a computer) or from a *sensor*, a device that measures some quantity in the environment.

A sensor will measure *digital* or *analog* phenomena. The simplest example of a digital sensor is a simple switch. A keyboard is, of course, an example of a digital sensor. Analog sensors can measures quantities such as position, temperature, pressure etc. Examples of analog sensors include joysticks (position) and microphones (sound). Each measurement is called a sample.

Before a measurement made by an analog sensor can be processed it must be converted to digital form as a binary number, in a process called *digitising*, by a device called an *analog to digital converter (ADC)*. This binary number will probably need to be *conditioned* or *scaled* in order to produce the kind of measurement sought. For example, the hardware specification for a temperature sensor and its ADC might tell us that the output of the ADC is 10000_{10} at 20 degrees Celsius and that the output goes down 10 for each degree rise in temperature. Can you figure out a formula to convert the output of the ADC into degrees Celsius?

The input data stream

Whatever the source of input data, it can be thought of as a stream of binary digits or bits.

… 1011010101010101010101011101000101010110100010101010 …

Often the stream of bits will be divided into some sort of regular chunks such as eight-bit bytes. Note that the chunks might not be eight-bits. Remember, for example, that ASCII is a seven-bit code and some old printers work with seven-bit chunks. Teletypewriters once worked with the five-bit Baudot code.

So a very general algorithm for processing an input data stream would be something like this:

```
BEGIN Process data stream
  WHILE there is more data to be processed
      Get the next chunk of data
      Process the chunk of data
  END WHILE
END
```

The input data stream can contain several different kinds of data:

- *Header information*—data at the beginning of a block of data, which might include information about the kind of data, the device that generated the data, the date and time the data was generated, error checking data and, commonly, the amount of data to follow.
- *Data body*—the actual data to be processed, which might include control code data. Control code data is meant to control aspects of the hardware (such as carriage returns) and will need to be separated from the data during processing. Control data is sometimes included through the use of escape sequences. An escape sequence uses a special control code that means that a certain number of data chunks that follow contain control data (see the dot matrix printer case study on p. 307).
- *Trailer information*—data at the end of a block of data, which could include error checking data and markers to indicate the end of the block of data.

The output data stream

Just like the input stream, the output stream can be thought of as a stream of bits grouped into chunks such as eight-bit bytes. As noted previously, the output stream from one system can become the input stream for another. Again, just like the input stream, the output stream can consist of a header, body and trailer data, control codes, escape sequences and data bits.

The output data stream is often designed to control a hardware device (such as a printer or an industrial robot arm). The processor and the device being controlled form a *control system*. A control system can be *open* or *closed*. In an open control system data moves only from the processor to the device being controlled, as in Figure 10.21a. A closed loop control system has a *feedback* loop from sensors which detect information about the effect of the device being controlled, as in Figure 10.21b.

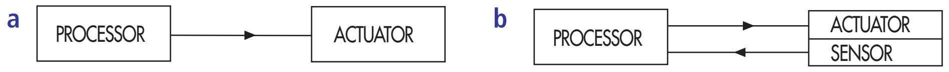


Figure 10.21 **a** An open control system. **b** A closed control system.

For example, a system that controls a room heater might operate simply by turning the heater on and off at predetermined times. This would be an open system. A closed heating system would include a temperature sensing device (a thermometer) which feeds data about the temperature of the room back to the system controlling the heater. Devices that have an effect on the real world, such as heaters and electric motors, are called *effectors* or *actuators*.

Electric motors are a very common device in controlled systems. How many devices common to homes and schools can you think of that include an electric motor? Some electric motors are operated as part of closed systems, with sensors that detect the position or the speed of the motor being fed back to the controlling sytem.

There are kinds of electric motor that can be operated in an open control system. Servo motors move through a defined angle proportional to the frequency of a controlling pulse; these are commonly used in model aeroplanes. Stepper motors move through an exact angle for each controlling pulse they receive; these are used in many computer devices, such as printers and disk drives. For example, the system that controls the print head of an inkjet printer can control exactly how far the print head travels, by sending a precise number of pulses to the stepper motor which drives the head backwards and forwards. This open control system would normally be used in conjuction with feedback sensors that detect when the print head is at the extreme left- and right-hand limits of its travel.

Output hardware devices vary in the amount of processing they do. People often talk loosely about the amount of 'intelligence' a device has. For example, a room heater can simply be on or off; it does not do any processing of its data stream. An old-fashioned teletypewriter used hardware to detect which input code corresponded to which character (or control function); again it could not be said to do any real processing of its input stream. Since the development of microprocessors, it has been possible to design cheap devices that perform some processing of their input; in fact software to control is often cheaper than the equivalent hardware. Modern dot matrix printers can be used in flexible ways because of the processors they contain. PostScript laser printers are fully fledged computer systems with their own programming language. (If your computer room contains a PostScript printer, it is probably the most powerful computer in the room.)

It is part of the job of the designer of software for a hardware device to work with engineers in determining which functions of the device are best done by hardware and which are best done by software.

Case study ASCII

Earlier in the chapter the ASCII code for representing text was discussed. The ASCII codes from 000 0000 to 001 1111 (0 to 31_{10}) and 111 1111 (127_{10}) are control characters. The code was designed to be compatible with punched tape. Where there is a 1, a hole is punched; for a 0, there is no hole. You would not want the case where there are no holes punched (000 0000) to stand for anything, so 000 0000 is said to stand for the NULL character. Similarly, if you made a mistake on a tape puncher, you could backspace and delete the erroneous character by punching all seven holes (111 1111), so 111 1111 = 127 is a deleted (or DEL) character.

Some of the most interesting control codes are the so-called format effectors. These affect the way the carriage on an old typewriter or teletype moves, or the way the cursor moves on a screen, as shown below.

- FE0 BS: backspace. Move the print head or the cursor one space to the left.
- FE1 HT: horizontal tab. Move the print head or cursor to the next tab position. This code corresponds to the tab key on the keyboard ('tab' is short for 'tabulating'). The tab character is often used to separate items of text, such as the fields in a database.
- FE2 LF: line feed. Move the cursor down one line. See the discussion about LF and CR below.
- FE3 VT: vertical tab. Like HT, only the cursor moves vertically.
- FE4 FF: form feed. Move to the next page; that is, feed one form through the printer.
- FE5 CR: carriage return. Move the cursor or print head back to the start of the line. See the discussion about LF and CR below.

Line feeds and carriage returns

You can see that there is no single format effector that causes the print head or cursor to move down to a new line. You need a combination of a line feed (move down one line) and a carriage return (move to the beginning of the line). Because of this, some systems (computers, printers, terminals) interpret CR to mean both CR and LF. This can get confusing if two (or more) connected pieces of equipment (say a computer and a printer) both interpret CR as meaning CR followed by LF. In this case you end up with two line feeds and everything ends up double spaced (or worse). Conversely, if no part of the system puts in a line feed, everything gets printed on the same line.

When text is stored using the ASCII code in a computer's memory or in a disk file, one byte is used for each character. There needs to be some way of marking the ends of lines. Do you use CR and trust some other part of the system to put in LF or do you use both CR and LF? The sad fact is that it depends on what kind of computer you are using: the convention on the Macintosh is to use just CR and on IBM compatibles to use both LF and CR.

Example

Design an algorithm to control a device that displays text from ASCII input. The device responds to the CR and LF control codes. The NULL (0) control code marks the end of the data input stream. All other control codes should be filtered out.

```
BEGIN Display ASCII input
  Get first character
  WHILE character is not NULL
      IF character <= 31 or character = DEL THEN
       It's a control character
       SELECT CASE character
             CASE CR: move cursor to start of line
             CASE LF: move cursor down one line
             CASE ELSE
                 Ignore this character
       END SELECT
      ELSE
       Display character
      END IF
      Get next character
  END WHILE
END
```

Questions

1 a How could this code be modified to also handle the BS control code?

b How could the code be modified to count the characters in the input?

c How could the code be modified to count the characters printed.

d How could the algorithm be changed if, instead of marking the end of the input with a NULL, the input included a header consisting of two bytes containing the number of characters to follow?

e How could the code be modified so that a CR or an LF control code is interpreted as 'Make a new line'; that is, either CR or LF is counted as being both a CR and an LF?

f How could the code be modified so that after every 80 characters the display moves to a new line.

2 Getting harder. How could the code be modified so that the HT (TAB) control code will cause some spaces to be printed so that the cursor moves across to the next position which is a multiple of 10 characters from the left of the display?

Case study A simple dot matrix printer

This case study describes the Acme rr80, a fictitious dot matrix printer designed to illustrate a number of ways that control characters and escape sequences are used. The printer's print head has eight pins arranged vertically which print onto the paper by striking through an inked ribbon as the print head moves across the page. The printer has an on-board processor that generates the dot patterns for characters from a number of fonts stored in ROM.

The input to the Acme rr80 is a stream of bytes, where all eight bits are significant. Each byte is printed as a character from the standard Latin 1 encoding unless the byte is a control code (hexadecimal 00 to 1F). The control codes are ignored except for the following:

- The format effectors have their standard effects. LF is a true line feed and CR is a true carriage return; that is, a CR, LF sequence is required for a new line.

- ESC (hexadecimal 1B, decimal 27) begins a sequence of one or more bytes affecting the control of the printer. Only a small selection of these 'escape' sequences is described here.
  - ESC F n sets the font to font number n where 1 = Times, 2 = Helvetica and 3 = Courier. For example, noting that the ASCII for the letter F is hexadecimal 46, the hexadecimal sequence 1B 46 03 would set the font to Courier.
  - ESC S n sets the print style to a style determined by n. The parameter n is calculated by adding values for the styles required. A value of 0 for n (decimal) will give plain text.

1	Bold
2	Italic
4	Underline
8	Expanded
16	Condensed
32	Shadowed
64	Outlined

  - ESC H n1 n2 n3 … NULL sets horizontal tab positions across the page. For example, the hexadecimal sequence 1B 48 08 10 18 20 00 sets tab positions 8, 16, 24 and 32 character positions from the left margin. (Note the 00 (NULL) at the end.)
  - ESC L n sets the number of lines per inch; that is, the sequence sets the amount by which each line feed feeds the paper. Common values for n are 10 (hex 0A) and 12 (hex 0C).
  - ESC G n1 n2 d1 d2 d3 … switches to a bit-mapped graphics mode where each data value d1, d2, d3 … prints one vertical column of eight dots, one dot for each of the eight pins in the print head, with the least most significant bit being the top dot. The parameters n1 and n2 form a 16-bit number defining the number of data values to follow, with the most significant byte first. This allows the printer to print any bit-mapped image by scanning successive rows of eight dots. Note that the printer must first be set to 12 lines per inch (see ESC L), or there will be a gap between each scanned row.

Notes

1 There are three kinds of escape sequence here: those with a fixed number of parameters (e.g. ESC F), those with a variable number of parameters with a delimiter or guard value to mark the end (e.g. ESC H), and those with a variable number of parameters where the number of data values is defined by the parameters (e.g. ESC G).

2 Modern GUIs such as Mac-OS and Microsoft Windows rarely use the text-printing capabilities of dot matrix printers. The printer drivers in these systems render each page to be printed as a bit map which is sent to the printer using something like the ESC G sequence.

Questions

1 In hexadecimal, what sequence of bytes would print these lines, in plain text?
Paris in the spring.

2 What would be printed by the following?
52 75 64 65 20 77 6F 72 64 08 08 08 08 08
08 08 08 08 58 58 58 58 58 58 58 58 58

3 What sequence of bytes would print the word 'Title' in bold, italic, underlined Helvetica.

4 What sequence of bytes would be required to set tab stops every 5 characters up to and including the 40th character position.

5 What sequence of bytes would print a 10 by 20 dot rectangle?.

Exercise 10.5

1 a Name and describe two different kinds of digital sensor.
  b Name and describe two different kinds of analog sensor.
  c What information might be included in the header of a data stream?
  d What kind of information might be contained in the trailer of a data stream?
  e Describe the purpose of a control code in a data stream.
  f What is the purpose of the carriage return (CR) and line-feed (LF) control codes in an ASCII data stream?
  g Describe the function of a stepper motor.
  h What is the difference between an open and a closed control system?
  i Describe an escape sequence in a data stream.
  j Would a printer be likely to contain any sensors? What would they sense?

2 Investigate the kind of data stream that is available from the keyboard on your computer? Is it possible to tell whether a key is being held down? Is it possible to tell which keys are being held down if more than one is being pressed simultaneously? Can you tell if the control, alt/option, shift keys are being pressed? Is the data code corresponding to a key the same as the ASCII code for the letter on that key? The place to find answers to these questions might be the help system for the programming language you use.

3 See if you can find at home or at school the manual for a modem (or search for documentation for the 'Hayes AT command set' on the Internet). What sequence of characters would be sent to the modem to get it to 'pick up the phone' and dial a number? What sequence of characters would be sent to the modem to get it to 'hang up' the phone?

4 A simple robot has the following commands:
- F *n* means move forward *n* steps. F means the ASCII value for F, and *n* is an eight-bit twos complement binary number (that is, *n* can be negative, so the robot can move backwards).
- R *n* means turn right (clockwise) through *n* degrees. Again, R stands for the ASCII for R and *n* is an eight-bit twos complement binary number. Negative values of *n* mean turn left (anti-clockwise).

  a What is the greatest number of steps that the robot can move forwards in one command?
  b What sequence of commands, written as binary numbers, would move the robot clockwise around a square with sides 20 steps long?
  c What sequence of commands, written as binary numbers, would move the robot anti-clockwise around a square with sides 20 steps long?
  d What sequence of commands causes the robot to rotate through one full revolution (360 degrees) on the spot?

5 As a class exercise, make a list of all the hardware devices you can think of that are in your school and that can be controlled by a computer or contain their own processor. Select one of these devices and answer the following questions:
  a What is its function?
  b What sensors does it have?
  c Can it be described as an open or a closed control system?
  d Does it have a processor that can be programmed?

Review exercises

1 **a** In a binary system there are __________ possible values.

b The radix of hexadecimal numbers is __________.

c A closed loop control system has __________.

d The middle digit in the binary number 1001001 stands for the __________ power of 2.

e The ASCII for a star is the answer to everything. What is it?

f What is the ASCII code for the numeral 7?

g What is the twos complement of 11001100?

h Write decimal –7 as a binary number in sign plus modulus form.

i What is the exact decimal equivalent of a kilobyte?

j What is the ones complement of 1111 1111?

2 Convert these decimal numbers to binary and hexadecimal.

a 10 **b** 22

c 75 **d** 921

3 Convert these numbers to decimal.

a 1101_2 **b** $1101\ 0110_2$

c 275_8 **d** AB_{16}

4 Perform these calculations (in binary). Show all your working.

a 1101 1100 + 0011 0101

b 0100 1100 – 0001 0011

c 0001 1010 × 0000 0100 (look for shortcut)

d 0010 1000 ÷ 0000 0100 (look for shortcut)

e 0010 1100 × 0000 1101

f 0001 1000 ÷ 0000 0100

5 **a** A binary logic system has two values, usually called __________ and __________.

b How many flip-flops are needed to store a byte?

c How many half-adders are needed to add two binary numbers each with four digits?

d The common element used in the manufacture of integrated circuits is __________?

e What is the output of an AND gate if one of its inputs is false?

f What is the output of an OR gate if one of its inputs is false?

g Why is a flip-flop called a bistable device?

h How many different combinations of inputs are there to a gate that has three inputs?

i If the input to a NOT gate is true, what is the output?

j If 573 NOT gates are connected in series, with the output of the first becoming the input of the second and so on, what will be the output of the 573rd if the input to the first is false?

6 Draw the logic circuits that will give the outputs described by the following truth tables and description.

a

Inputs		Output
A	B	
0	0	0
0	1	0
1	0	0
1	1	1

b

Inputs		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

c

Inputs		Output
A	B	
0	0	1
0	1	0
1	0	0
1	1	1

d A logic circuit with three inputs where the output is true if any of the inputs is false.

7 **a** Show how logic gates can be used to build a device that can store one bit of data.

b Describe a binary half-adder. Show how half-adders can be combined to construct a full-adder.

8 Describe the kinds of data that might make up a data stream (input or output), including a header, data, trailer information and control codes.

Chapter summary

- Binary numbers are used to represent data in computer systems because there are many things that can be in one of two states.
- Binary numbers work in the same way as decimal numbers except that they are based on powers of two instead of powers of ten.
- Computer programmers use octal or hexadecimal numbers because eight and sixteen are powers of two.
- In a computer system, numbers can be represented directly as binary numbers or in a character code such as ASCII.
- The addition of binary numbers can be carried out using an algorithm just like the familiar algorithm for decimal numbers. The usual way of subtracting a binary number is to add its twos complement.
- The multiplication of binary numbers can be done on paper using the familiar method. In a computer an algorithm involving shifting and adding is used. Division uses an algorithm involving shifting and subtracting.
- Electronic circuits to process binary data can be built up using simple logic gates such as NOT, AND and OR gates. The inputs and outputs of a logic circuit can be represented using a truth table.
- A logic circuit to perform addition of two binary digits with carry is called a full-adder. A sub-circuit of the full-adder is the half-adder which adds two bits without carry.
- Data can be stored in logic circuits called flip-flops or bistable devices.
- Very complex circuits can be built up into single packages called integrated circuits, some of which contain the equivalent of millions of simple logic gates. Like other design tasks, the design of logical circuits proceeds in a cycle of steps from identification of the problem through to evaluation of the design.
- A system processes data from other sources such as another processor or a sensor— a device that measures some quantity in the environment.
- Sensors may be digital or analog. The signal from an analog sensor must be converted to digital form by an analog to digital converter (ADC).
- The input may be thought of as a stream of bits. The data stream can include header, body and trailer information.
- A system can control other hardware through an output data stream that similarly can include header, body and trailer information. Hardware devices that affect the outside world are called effectors or actuators.
- Input and output data streams can include data characters that are to be displayed or processed and control characters that control hardware features (such as carriage returns). Escape sequences in the data stream can switch between kinds of data.
- When a system controls an output device but data about the effect is not fed back to the controlling processor, an open control system is formed. When there is a sensor that feeds back to the controlling processor, a closed control system is formed.

Appendix: ASCII code

Character	Description	Decimal	Hexadecimal	ASCII code
NUL	Null	0	0	0000000
SOH	Start of heading	1	1	0000001
STX	Start of text	2	2	0000010
ETX	End of text	3	3	0000011
EOT	End of transmission	4	4	0000100
ENQ	Enquiry	5	5	0000101
ACK	Acknowledge	6	6	0000110
BEL	Bell	7	7	0000111
BS	Backspace	8	8	0001000
HT	Horizontal tab	9	9	0001001
LF	Line feed	10	A	0001010
VT	Vertical tab	11	B	0001011
FF	Form feed	12	C	0001100
CR	Carriage return	13	D	0001101
SO	Shift out	14	E	0001110
SI	Shift in	15	F	0001111
DLE	Data link escape	16	10	0010000
DC1	Device control 1	17	11	0010001
DC2	Device control 2	18	12	0010010
DC3	Device control 3	19	13	0010011
DC4	Device control 4	20	14	0010100
NAK	Negative acknowledge	21	15	0010101
SYN	Synchronous idle	22	16	0010110
ETB	End of trans block	23	17	0010111
CAN	Cancel	24	18	0011000
EM	End of medium	25	19	0011001
SUB	Substitute	26	1A	0011010
ESC	Escape	27	1B	0011011
FS	File separator	28	1C	0011100
GS	Group separator	29	1D	0011101
RS	Record separator	30	1E	0011110
US	Unit separator	31	1F	0011111
Space		32	20	0100000
!	Exclamation mark	33	21	0100001
"	Double quote	34	22	0100010
#	Hash	35	23	0100011
$	Dollar	36	24	0100100
%	Percent	37	25	0100101
&	Ampersand	38	26	0100110
'	Quote	39	27	0100111

(	Open parenthesis	40	28	0101000
)	Close parenthesis	41	29	0101001
*	Asterisk	42	2A	0101010
+	Plus	43	2B	0101011
,	Comma	44	2C	0101100
-	Minus	45	2D	0101101
.	Full stop	46	2E	0101110
/	Oblique stroke	47	2F	0101111
0		48	30	0110000
1		49	31	0110001
2		50	32	0110010
3		51	33	0110011
4		52	34	0110100
5		53	35	0110101
6		54	36	0110110
7		55	37	0110111
8		56	38	0110000
9		57	39	0111001
:	Colon	58	3A	0111010
;	Semicolon	59	3B	0111011
<	Less than	60	3C	0111100
=	Equals	61	3D	0111101
>	Greater than	62	3E	0111110
?	Question mark	63	3F	0111111
@	Commercial at	64	40	1000000
A		65	41	1000001
B		66	42	1000010
C		67	43	1000011
D		68	44	1000100
E		69	45	1000101
F		70	46	1000110
G		71	47	1000111
H		72	48	1001000
I		73	49	1001001
J		74	4A	1001010
K		75	4B	1001011
L		76	4C	1001100
M		77	4D	1001101
N		78	4E	1001110
O		79	4F	1001111
P		80	50	1010000
Q		81	51	1010001
R		82	52	1010010
S		83	53	1010011

T		84	54	1010100
U		85	55	1010101
V		86	56	1010110
W		87	57	1010111
X		88	58	1011000
Y		89	59	1011001
Z		90	5A	1011010
[	Open square bracket	91	5B	1011011
\	Backslash	92	5C	1011100
]	Close square bracket	93	5D	1011101
^	Caret	94	5E	1011110
_	Underscore	95	5F	1011111
`	Back quote	96	60	1100000
a		97	61	1100001
b		98	62	1100010
c		99	63	1100011
d		100	64	1100100
e		101	65	1100101
f		102	66	1100110
g		103	67	1100111
h		104	68	1101000
i		105	69	1101001
j		106	6A	1101010
k		107	6B	1101011
l		108	6C	1101100
m		109	6D	1101101
n		110	6E	1101110
o		111	6F	1101111
p		112	70	1110000
q		113	71	1110001
r		114	72	1110010
s		115	73	1110011
t		116	74	1110100
u		117	75	1110101
v		118	76	1110110
w		119	77	1110111
x		120	78	1111000
y		121	79	1111001
z		122	7A	1111010
{	Open curly bracket	123	7B	1111011
|	Vertical bar	124	7C	1111100
}	Close curly bracket	125	7D	1111101
~	Tilde	126	7E	1111110
DEL	Delete	127	7F	1111111

Glossary

actuator (effector) a device under the control of a signal from a controller (e.g. a motor, a light, a loudspeaker).

algorithm a series of steps which, when performed correctly, will solve a problem in a finite time.

alphanumeric data data consisting of letters of the alphabet and the numbers.

AND gate a logic circuit with two inputs and one output. The output is true if, and only if, both inputs are true.

application software software that performs a specific task.

array a *structured data type* containing a number of related data items, each having the same data type.

ASCII stands for American Standard Code for Information Interchange, which is the common name for the standard seven-bit code for representing text. See *Australian Standard Coded Character Set (AS 1776–1980.)*

Australian Standard Coded Character Set (AS 1776–1980) the Australian standard code for representing text (commonly known as ASCII).

binary digit (bit) a digit that can have one of two values, zero or one.

binary numbers numbers written in a system based on two.

binary selection a *control structure* in which a choice of two paths is presented. The path executed depends on the result of a condition.

binary system a system that can be in one of two states; for example a switch can be on or off.

binary-coded decimal (BCD) numbers numbers represented using four bits (one nybble) for each decimal digit.

bistable devices devices that are stable in one of two possible states.

bit a single binary digit. It is the smallest unit of storage in a digital computer.

BNF (Backus-Naur form) a text-based method of stating the rules of a language. See also *EBNF* and *syntax structure diagram*.

Boolean data type a data type in which only two possibilities, usually either true or false, are represented by a variable.

breakpoint a place 'marked' in a program where execution of the program is suspended so that the values of variables can be examined. Breakpoints are usually used only during debugging of a program.

buffer a logic gate with one input and one output. The output is the same as the input.

byte a group of eight bits.

CAD software software that is used as an aid in the design of a product. There are many different kinds of CAD software. The development of integrated circuits (ICs) has only been possible through the development of suitable CAD programs.

called a subprogram is *called* when control passes to that subprogram from the main program.

calling module the module from which a call to another module is made.

central processing unit (CPU) the CPU retrieves, decodes, interprets and executes instructions.

character the smallest unit of data normally handled by people.

character data type a *simple data type* in which only one coded character can be represented by the variable.

command based interface a *human–computer interface* in which the person has to type in commands in order to manipulate data.

command-line interface an alternative term used to describe a *command-based interface*.

compilation complete translation of the source code to object code.

compound statements a statement in a programming language which combines a number of instructions, each of which is a simple statement.

concatenation joining two strings together.

configuration management the management of software resources.

constant a value which cannot be changed during the execution of a program.

control one of the logical elements of a computer system. The control element of a computer system coordinates the processes that are carried out.

control codes codes as part of a system such as ASCII, which do not print visible characters but which control equipment (e.g. carriage return).

control structures a term which describes the three basic structures of an algorithm (*sequence*, *selection* and *repetition*).

data validation a check made by the computer that data is within allowable limits for processing.

data verification the manual process of checking that data items have been entered correctly by comparing the entered data with the source data.

database management systems a software system which allows a database to be created, maintained and accessed.

dataflow diagram a representation of the paths of data through the system.

decimal numbers numbers written in a system based on ten.

decompilation conversion of executable machine code to assembler language.

decrement to decrease the value of a variable by 1.

desk check manually checking the logic of an algorithm by using test data.

direct access a method of accessing data where a record can be accessed without having to access any of the previous records. Also known as *random access*.

direct cut over a complete and immediate conversion to the new system.

double precision the use of a greater than normal number of bytes to store a numerical value, allowing a greater degree of accuracy.

EBCDIC (extended binary coded decimal interchange code) a binary coding for characters which uses eight bits to represent each character. See also *ASCII*.

EBNF (extended Backus-Naur form) a text based method of stating the rules of a language. See also *BNF* and *syntax structure diagrams*.

effector (actuator) a device under the control of a signal from a controller (e.g. a motor, a light, a loudspeaker).

encapsulation hiding of processing details within an object.

end of file (EOF) mark a pattern of bits which represents the character used to indicate the end of a string of text. It is usually used to show the end of a file stored on an external storage medium such as a magnetic disk.

evolutionary prototyping a method of *prototyping* in which the prototype is developed into the final software solution to the problem.

feedback the return of some part of output to be used as input in a closed loop, computer controlled system.

field one data item within a record data structure.

file a block of data which may have been written to a storage device.

flip-flop basic characteristic is that it can be in only one of two possible states at any given time and it will remain in that state until commanded to change states. Because flip-flops stay in one of two states they are called bistable devices.

floating point a way of representing fractional numbers using a mantissa and an exponent, as in scientific notation.

full-adder a logic circuit with three inputs and two outputs. The outputs show the sum of the three inputs (one of which might be the carry from a previous sum).

function a predefined set of operations which returns a value.

graphical user interface (GUI) a human-computer interface which employs icons and menus to assist the user to navigate through the choices of a program.

guarded loop a loop in which the decision is placed at the start of the loop. In pseudocode, it is the WHILE......ENDWHILE structure.

half-adder a logic circuit that has two inputs and two outputs. One output is the sum of the two inputs and the other is the carry generated by the sum.

hexadecimal a counting system based on sixteen. We use the characters 0 to 9 and A to F to represent the sixteen digits needed for the hexadecimal system.

identifier a name given to a *constant*, *variable*, *function* or *subroutine* of a program.

increment to increase the value of a variable by 1.

incremental compilation interpretation of the program with common routines being compiled.

index the value which represents the position of a data item in an array.

input the process of transferring data into a computer system from outside by means of a peripheral device.

input/output table a table of test data that lists the test data items and the expected outputs.

integer data type a data type used to represent positive and negative whole numbers.

integrated circuit (IC) the entire circuit for a complex function can be made into a single block.

intellectual property a work resulting from creative activity.

internal documentation documentation included in the source code, consisting of *intrinsic documentation* and remarks (also called comments).

interpretation translation and execution of a program line by line.

intrinsic documentation documentation 'built into' the source code. The main type of intrinsic documentation is the appropriate choice of *identifiers*, the use of indentation to show program modules may also count as intrinsic documentation as it makes the logic of the program more clear.

IPO (Input Processing Output) chart an IPO chart tabulates the inputs, processes and outputs required for a system.

iteration looping through a process a number of times. See also repetition.

least significant bit the bit in a binary string that has the smallest value. It is usually the bit at the extreme right of a byte, or group of bytes.

linear search a search which progresses, one element at a time, from the first indexed element of an array towards the last.

logic a system in which variables have one of two values (true or false).

logic gate a circuit that has one or more inputs and one or more outputs and where the inputs and outputs can take one of two values. The rule for converting the inputs to the outputs can be shown with a truth table.

loop an alternative term for a *repetition* or *iteration*.

metalanguage a method of describing the syntax of a language.

module a part of a program, such as a subprogram or function, which performs a specific task. A module will pass data to and/or accept data from other parts of the program.

most significant bit the bit in a binary string that has the greatest value. It is usually the leftmost bit of a byte.

multiway selection a control structure in which a choice is made from a number of alternatives. The choice is based on the value of an expression.

NAND gate a logic circuit with two inputs and one output. The output is true if, and only if, both inputs are not true (NOT AND).

nibble a four bit binary string.

nonterminal symbols elements of a language that are defined elsewhere in the language description. See also *terminal symbols*.

NOR gate a logic circuit with two inputs and one output. The output is true if neither of the two inputs is true (NOT OR).

NOT gate a logic gate with one input and one output. The output has the opposite value to the input.

nybble an alternative spelling for *nibble*.

octal a counting system based on eight only the digits 0 to 7 are used to represent numbers. See also *binary* and *hexadecimal*.

ones modulus a method of representing integers (positive and negative numbers) in the binary number system.

operating system the software that manages the resources of a computer.

OR gate a logic circuit with two inputs and one output. The output is true if either of the two inputs is true.

output the process of transferring data from a computer system to the outside by means of a peripheral device.

outsourcing using outside contractors for a development task.

overflow a condition in which the result of an operation is too large for the storage allocated.

parallel conversion the full use of both the new and old systems for a period of time.

phased conversion gradual implementation of the new system.

pilot conversion complete installation of the new system with only some of the tasks being performed by it, the old system performing the rest.

pixel the smallest element of a screen display that can be displayed.

post-test repetition an iteration in which the termination test is after the body of the loop. In pseudocode a post-test repetition is identified as a REPEAT.......UNTIL construction.

pre-test repetition an *iteration* in which the termination test is before the body of the loop. In pseudocode a pre-test repetition is identified as a WHILE.......ENDWHILE construction.

primary storage storage that is directly accessible to the CPU.

process an action (when the word process is used as a noun), to perform a set of instructions (when the word is used as a verb).

prototype a working model of an application which can be used to gather information. A prototype may be developed into the final application or it may be 'thrown away'.

radix the number on which a number system is based.

random access a method of accessing data where a record can be accessed without having to access any of the previous records. Also known as *direct access*.

random access memory (RAM) *primary storage* which can be written to as well as read. See also *read only memory*.

read only memory (ROM) *primary storage* which can only be read. See also *random access memory*.

record a collection of related facts. A record is stored as one or more *fields*.

recursion a definition which incorporates itself.

register a temporary storage location within the CPU.

reliability the ability of software to perform without failure.

repetition an algorithm structure in which a sequence of steps may be executed a number of times.

screen buffer an area of *primary storage* which is used to store the data which represents the image on a screen.

secondary storage storage which is not directly accessible by the CPU.

selection an algorithm control structure which presents two or more options, the choice of which depends upon the result of a test.

sensor a device that measures some aspect of the world.

sentinel value a value used to 'mark' the end of a data list.

sequence an algorithm control structure which consists of a number of steps one after the other.

sequential file a file structure in which, to reach one record, each of the preceding records has to be passed over.

sign plus modulus a method of representing integers by using one bit for the sign (positive or negative).

simple data type a data type that may be applied to one data element.

single precision the standard representation of a simple numerical data type within a particular programming language.

stack a part of main memory used to store the location of the instruction to return to after a subprogram has been executed.

standard constructs the general term which describes the basic elements of an algorithm; *sequence*, *selection* and *repetition*.

statement a single step as written in a programming language.

stepwise refinement a process in which a problem is broken down into smaller parts until it can be easily solved.

storage a device or medium that can be used to hold data.

string data type a *simple data type* consisting of a number of *characters*.

structured data type a data type which is used to represent a number of related data elements as one data item.

stub a small *module* representing a part of the program which is still to be written.

subroutine a part of a program which performs a specific task.

syntax the set of rules that govern the way in which the elements of a language can be combined to form a *statement*.

syntax graphs a pictorial method of illustrating the syntax of a language. Also known as a *syntax structure diagram*.

syntax structure diagrams a pictorial method of illustrating the syntax of a language. Also known as a *syntax graph*.

system software the files and resources needed by a computer system in order to allow it to run properly. System software includes the *operating system* and *utility software*.

terminal symbols individual characters, or strings of characters, which are used in the definition of a syntax structure.

test data data elements designed to test the operation of an *algorithm* or program.

text data that can be written out using normal letters of the alphabet, numerals and punctuation marks.

top-down design a design approach in which a problem is broken down into a number of smaller and easier to solve problems.

tracing the process of following the execution path of a running program in order to identify the source of an error.

transistor an electronic component that can be used as the basis of circuits to build logic gates (among many other applications).

truncation the 'loss' of accuracy caused by a limit to the way that results of operations can be stored.

truth table a table that lists all the possible combinations of inputs for a logic gate with the corresponding outputs.

twos complement a method of using a fixed length binary string of digits to represent both positive and negative integers.

twos modulus a method of representing integers (positive and negative numbers) in the binary number system.

unguarded loop a loop whose body must be executed at least once each time it is reached. Also known as a *post-test repetition*.

user interface the link between the user and the computer program. The most common user interfaces use screens, keyboards and mice.

utility software programs that perform management tasks such as formatting disks, duplicating files, virus protection, etc.

validation comparison of the solution with the design specification.

variable a name used within the code of a program to reference a stored data element

variable declaration a statement in a programming language that indicates the type of data that a variable will be used to store.

verification ensuring that the software performs its functions correctly.

virtual memory a technique which uses secondary storage in the place of primary storage so that the computer appears to have more main memory than it really does.

word length the maximum number of bits that can be processed at one time by a CPU.

XOR gate a logic circuit with two inputs and one output. The output is true if either of the two inputs is true but not if both are true (exclusive OR—sometimes written EOR).

Index

Acknowledgments

The authors and publisher would like to thank the following for permission to reproduce copyright material in this book:

The Age, pp. 4 top, 13 top, 50; Australian Picture Library, pp. 4 bottom, 13 bottom; Bill Bachman, pp. 24; Borland, p. 11; ClarisImpact, pp. 176, 177; Coo-ee Picture Library, p. 49; Malcolm Cross, p. 34; Great Southern Stock, pp. 202, 204; IBM, p. 26; Dale Mann/Retrospect Photojournalism, pp. 14, 21, 34, 216; Northside Photos, p. 28; United Feature Syndicate Inc., p. 228, 229, 241; Universal Press Syndicate copyright 1993 G. B. Trudeau, p. 193 top; Valve Software, p. 66.

Every effort has been made to trace and acknowledge copyright. The authors and publisher would welcome any information from people who believe they own copyright to material in this book.

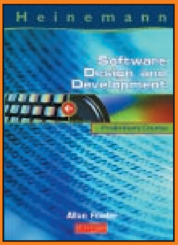
Heinemann Software Design and Development: HSC Course is the second in a series written to provide comprehensive coverage of the new Stage 6 Software Design and Development syllabus in NSW. Experienced author, Allan Fowler, and a team of leading secondary and tertiary contributors, have combined their expertise in this systematic guide for students. The book covers the main aspects of software design and development. This includes analysing the problem, planning, creating, testing and documenting the solution as well as the associated social and ethical considerations. It uses a wide range of computer languages so that students learn to use the most appropriate to develop each solution.

Key features

- matched exactly to the new Stage 6 HSC Course
- list of outcomes at beginning of each chapter
- end-of-chapter summary to reinforce learning
- chapter review exercises
- team exercises where appropriate
- skills development through graded activities
- questions to stimulate discussion on issues
- extensive glossary
- detailed index
- online support at *hi.com.au/softwaredesign*

About the author

Allan Fowler is an experienced Computing Studies teacher with particular expertise in programming languages and program development. He has many years' experience in teaching Computing Studies and is the author of the highly successful, *Heinemann Senior Computing Studies 3 Unit HSC Course* and *Heinemann Software Design and Development: Preliminary Course.*



ISBN 0-86462-513-8

9 780864 625137