UNIT 2

INTRODUCTION

Unit 2 of VCE Computing focuses on data and how computational, design and systems thinking skills are applied to support the creation of a range of solutions.

Throughout Unit 2, students will be required to apply the analysis, design, development and evaluation stages of the problem-solving methodology outlined in Unit 1. In Area of Study 1: Programming, students develop a range of knowledge and skills while using programming and scripting languages, and associated software, to create solutions. In Area of Study 2: Data analysis and visualisation, students expand on their knowledge of data and the various tools that are used to extract it, reduce its complexity and manipulate it to create clear, attractive and useful visualisations. In Area of Study 3: Data management, students use database management software to create a solution that applies all stages of the problem-solving methodology.

AREA OF STUDY 1: PROGRAMMING

Outcome 1

You are required to design working modules in response to solution requirements, and use a programming or scripting language to develop the modules.

To achieve this Outcome, the student will draw on key knowledge and key skills outlined in Area of Study 1.

AREA OF STUDY 2: DATA ANALYSIS AND VISUALISATION

Outcome 2

You are required to apply the problem-solving methodology and use appropriate software tools to extract relevant data and create a data visualisation that meets a specified user's needs.

To achieve this Outcome, the student will draw on key knowledge and key skills outlined in Area of Study 2.

AREA OF STUDY 3: DATA MANAGEMENT

Outcome 3

You are required to apply the problem-solving methodology to create a solution using database management software, and explain the personal benefits and risks of interacting with a database.

To achieve this Outcome, the student will draw on key knowledge and key skills outlined in Area of Study 3.

CHAPTER 6

PROGRAMMING

Key knowledge

After completing this chapter, you will be able to demonstrate knowledge of:

Data and information

- characteristics of data types and methods of representing and storing text, sound and images

Digital systems

- functions and capabilities of key hardware and software components of digital systems required for processing, storing and communicating data and information

Approaches to problem solving

- functional requirements of solutions
- methods for creating algorithms such as identifying the required output, the input needed to produce the output, and the processing steps necessary to achieve the transformation from a design to a solution
- suitable methods of representing solution designs such as data dictionaries, data structure diagrams, object descriptions and pseudocode
- characteristics of effective user interfaces, for example useability, accessibility, structure, visibility, legibility, consistency, tolerance, affordance
- techniques for manipulating data and information
- naming conventions for files and objects
- testing and debugging techniques, including construction of test data

Key skills

- interpret solution requirements
- select and use appropriate methods for expressing solution designs, including user interfaces
- apply techniques for manipulating data and information using a programming or scripting language
- devise meaningful naming conventions for files and objects
- apply testing techniques using appropriate test data.

For the student

This chapter relates to VCE Computing Unit 2, Area of Study 1: Programming. It introduces basic programming concepts such as software development tools, storage and control structures, the software development process, design tools, types of programming languages, and universal programming ideas such as pseudocode, modules, loops, debugging and testing. You will also be learning a specific programming language that you will use to develop a series of small programming tasks for Unit 2, Outcome 1.

For the teacher

This chapter introduces students to the universal theoretical concepts behind programming that are required for Unit 2, Outcome 1. It does not assume knowledge of, or use source code from any actual language. Instead, pseudocode is used. Little previous programming experience can be expected from many students, so it is important to introduce the chosen language to them early, and give them time to train up. Unit 2, Outcome 1 should consist of a few (perhaps five) small, independent tasks that include basic programming concepts, such as storage, logic, loops and calculations.

Information systems in programming

This chapter deals with Unit 2, Outcome 1 of Computing. Throughout this Area of Study, you will focus on using a programming or scripting language that is capable of supporting object-oriented programming (OOP) to create working software modules. Programming and scripting languages provide more flexibility than applications do, because you can insert specific instructions to create a purpose-designed solution.

Information systems are explained in Chapter 3.

The specific language that you study is flexible. You will also develop and hone your skills in interpreting solution requirements that come from your teacher, and in designing working modules. During this Area of Study, you will apply methods and techniques for completing a series of small discrete tasks or working modules that use features of a programming or scripting language, including predefined classes.

You will also apply knowledge and skills associated with the design and development stages of the problem-solving methodology (PSM).

Information systems comprise people, data, processes and digital systems. In the context of programming, the key parts are:

1. **people**, who interact with systems according to their needs, such as programmers, data entry operators, system managers, technicians and end users
2. **data**, which is composed of raw, unprocessed facts and figures, such as someone's date of birth, that is used as input to be processed into meaningful information as output, such as someone's age
3. **processes**, which are the manual and automated ways of achieving a result, such as a manual data backup or an automated hard disk error scan
4. **digital systems**, which are made up of the hardware and software needed to support programming and software use.

   Digital systems are made up of the following components.
   - Networks exchange data between computers.
   - Protocols are rules used to coordinate and standardise communication between devices.
   - Application architecture patterns are sets of principles used to provide a framework for structuring solutions to recurring problems; for example, **thin client** is the philosophy that, rather than use powerful computers it is better to use 'dumb' workstations connected to a powerful central computer that does all the processing work for them.
   - Software comes in three types: systems, applications and utilities.
   - Hardware is physical equipment for input, output, storage, processing and communication.

See Chapter 3 for more information on networks and protocols.

The following section will discuss in more detail the hardware and software components of digital systems.

Hardware

The physical components of digital systems are known as hardware. They include familiar items such as the monitor, mouse, hard disk drive (HDD), motherboard, graphics card, sound card and so on.

Hardware requires software instructions to control it; software requires hardware to carry out its instructions. They work together to form a useable digital system.

Hardware falls under a number of categories, including:

- **input devices**, which are instruments and peripherals, such as keyboards, that enable users to send data and commands to software and the operating system

- output devices, which are instruments and peripherals such as printers and monitors that display information from a computer in human-readable form
- processing hardware
- storage hardware
- communication hardware.

The following sections cover processing, storage and communication hardware in greater depth.

Processing hardware

The key element of programming hardware is the processing hardware – the digital processor that converts data into information and controls all of the other hardware in the system.

CPU

THINK ABOUT COMPUTING 6.1
Research the CPUs of a mobile phone, a laptop and a desktop gaming machine. How do they differ, and how is their performance measured?

The central processing unit (CPU) is often thought of as the 'brain' of a digital system and it handles most of a system's data manipulation. The CPU is helped by other processors, such as those in the video card, hard disk drives and audio controller chips. Major CPU designers include Intel, AMD, ARM and IBM.

Reduced instruction set computing (RISC) CPUs, such as ARM, have smaller instruction sets than **complex instruction set computing (CISC)** CPUs, such as Intel's i7. Being cheaper and smaller and therefore drawing less power and producing less heat makes CISC CPUs ideal for use in smartphones and tablets.

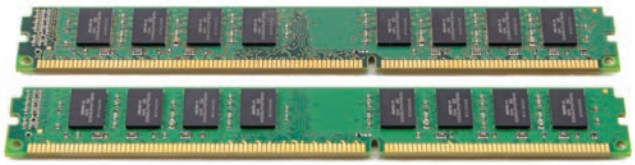
GPU

As CPUs get smaller, the laws of physics start to be bent. When electrons travel in time, or disappear and reappear elsewhere in the universe, CPUs will become unreliable. Quantum computing hopes to fix this problem.

The graphics processor unit (GPU) is a very fast and expensive processor specifically designed for high-speed image processing in graphics cards. Application software, such as Adobe Photoshop, video editors and 3D games, exploit GPU power to accelerate processor-intensive calculations.

Storage hardware

Storage hardware retains data and software for both immediate and later use. It comes in two main types: primary storage and secondary storage.



Shutterstock.com/ Jiggo_thekop

FIGURE 6.1 RAM modules

Primary storage

Primary storage is a computer's random-access memory (RAM). It has billions of storage locations in silicon chips. RAM stores instructions and values including variables, arrays and other storage structures when programs are running or being created. RAM chips are volatile because they lose their data when electricity is turned off. Dynamic RAM (DRAM) is used as the main memory in computers; high-speed (and expensive) static RAM (SRAM) is used in graphics cards and CPUs.

Secondary storage

Permanent secondary storage stores data, information and applications when they are not actively used. Secondary storage includes hard disk drives (HDD), solid state drives (SSD) and network-attached storage devices (NAS).

Hard disk drives (HDD) are aluminium disks densely crammed with magnetically recorded bits of 1 and 0. Spinning at up to 10 000 RPM, they store and retrieve data at incredible speed, with breathtaking accuracy and reliability. They are very cheap per megabyte of capacity, and

still the biggest, and most reliable long-term storage medium you can find. In 2015, a 4TB (approximately 4000GB) HDD cost approximately $200, which works out to about 20GB of storage per dollar.

Solid state drives (SSD) store data in non-volatile NAND RAM (similar to that used in Flash drives and SD cards). They have no motors to age and fail, run silently, start up instantly, consume less electricity, generate less heat, and may access data faster than a HDD. Unfortunately, NAND RAM eventually loses its ability to be written to, stores less data per square centimetre of storage space, and is expensive. In 2015, a 128GB SSD cost approximately $105, or around 1GB of storage per dollar.

A network-attached storage (NAS) device is a networked team of HDDs. Using a NAS offers more speed, capacity (e.g. 12TB), data protection (e.g. hot-swap disks), convenience and reliability than a simple USB hard disk alone.

Network-attached storage (NAS) is discussed in more detail in Chapter 3 on page 109.

TABLE 6.1 Storage units

Unit	Symbol	Equivalent to	
		RAM	Data storage
Byte	B	8 bits (1 or 0), the basic unit of storage	8 bits
Kilobyte	KB	1024 bytes	1000 bytes
Megabyte	MB	1024KB (roughly 1 million bytes – the size of two average novels)	1000KB
Gigabyte	GB	1024MB (PCs have gigabytes of RAM)	1000MB
Terabyte	TB	1024GB (hard disks have terabytes of storage)	1000GB
Petabyte	PB	1024TB	1 000 000GB (1×10^6GB)
Exabyte	EB	1024PB	1 000 000 000GB (1×10^9GB)
Zettabyte	ZB	1024EB	1 000 000 000 000GB (1×10^{12}GB)
Yottabyte	YB	1024ZB	1 000 000 000 000 000GB 1×10^{15}GB)

Storing 1YB would take 2 500 000 cubic metres of 64GB microSD cards – the equivalent of the volume of the Great Pyramid of Giza

THINK ABOUT COMPUTING 6.2
Get online Australian prices for various sizes of SSD and HDD. Graph their costs against their capacity.

Communication hardware

Communication hardware is used for sending and receiving data and information.

Ports are physical sockets or connectors that carry data between a computer and external devices, often referred to as peripherals. Universal serial bus (USB) is a standardised high-speed way to connect many devices, including Flash drives, printers, modems, keyboards, mouses, speakers and smartphones.

In the days before USB, many manufacturers invented their own type of port. Computers were jam-packed with ports to accommodate individual makes of modem, printer, mouse, keyboard, monitor, joystick, etc.

As a programmer of a high-level language you will not need to worry about directly controlling devices such as printers or disk drives. Your programming language will issue commands such as 'display this' or 'save this data' and the OS will negotiate with the hardware to fulfil your requests. The OS knows how to talk to hardware because each device comes with a software **driver**, which is like a dictionary that tells the OS the commands that the hardware understands. The OS gives a generic command, such as 'print this', and the driver translates the command into language that the specific piece of hardware understands. Hardware misbehaviour is often caused by using an incorrect or outdated driver.

Networking hardware – modems, routers, switches, cables and wi-fi are explained in Chapter 3.

Software

Software is used to control computing devices to process data. There are many types of software programs used to:

- calculate, such as spreadsheets
- store and organise data, such as databases
- entertain, such as games
- communicate, such as web browsers, email and instant messaging
- control devices, such as embedded software in TVs, toasters and car engines.

Hundreds of other programs exist. These pieces of software are all created by programmers.

Types of software

System software tools are used by a computer to manage hardware and run the user's programs; for example, the operating system (OS), device drives and communication protocols.

Applications are used to perform work or complete larger tasks. Popular examples of applications include Microsoft Word and Excel, Adobe Photoshop and Mozilla Firefox browser.

Utilities are usually small, single-purpose software tools that do a specific job or add functionality to an operating system. They include text editors, audio format converters and DVD burners.

The OS (operating system)

An OS such as Windows, Mac OSX, Linux or Android is system software that controls a computer's hardware and runs the user's application software. Operating systems are usually incompatible with one another, but they all perform similar functions.

Windows users can run *services.msc* to see the dozens of tasks the OS is managing in the background.

- Loading and saving data and programs
- Displaying output and printing
- Processing sound and music
- Allocating memory for user programs
- Watching the user's keyboard and mouse activity
- Controlling network and internet access
- Encrypting, decrypting, compressing and decompressing data
- Caching downloads
- Controlling user logins and maintaining security over accounts, files and access to resources
- Running background programs to keep the system working efficiently; for example, disk defragmenters, virus scanners and checking for upgrades

THINK ABOUT COMPUTING 6.3

Linux is a free, open source OS. Where is Linux used, and what benefits and drawbacks does it have compared with Windows and Mac OS?

Programming and scripting languages

There are more than 700 programming languages, but you will not have to learn them all.

Programming languages are used to give instructions to computer processors so they can calculate useful information or carry out tasks for humans. Whether your phone is playing an MP3, your car is turning on its anti-skid braking, or McDonald's is calculating staff wages, programming languages are needed.

Scripting languages conveniently store sequences of instructions that, alternatively, could be entered one at a time. Like human languages, there are many programming languages, each with distinctive grammar, punctuation and vocabulary. Most programming languages have special abilities or strengths that make them more useful than other languages for a particular task.

Professional programmers know a handful of languages and choose the best language for each job based on its strengths and weaknesses. Choosing the languages to learn is a big decision, but remember that learning one language makes it easier to learn others. The most popular programming languages include C (C++ or C#), Python, Java, JavaScript, Perl and PHP, SQL and Visual Basic.

- C, C++ or C# is used for writing low-level utilities and fast applications.
- Python is a scripting language used widely across the internet and to control devices.
- Java is used for server-side website programming and for Android apps.
- JavaScript is a client-side scripting language for websites.
- Perl and PHP are also used for websites.
- SQL, or structured query language, is a scripting language for database programming.
- Visual Basic is a good first programming language to learn and it is good for **prototypes**.

While programming languages may differ, they all do basically the same job: they control a digital system such as a computer, tablet or smartphone.

Programming languages differ in the amount of direct control they give over a computer's hardware and operating system. With a **high-level** language such as Visual Basic or Python, programmers avoid having to worry about complex details of the structure of actual disk files or where data is stored in memory. High-level languages are simpler to use, but lack the control of complex but more difficult to learn low-level languages. Conversely, a **low-level** language such as C or machine code requires more skill and knowledge from the programmer, but allows more direct control of the workings of a computer.

High- and low-level programming languages each have their uses. To write a simple alarm clock program, a high-level language is fine. To write a device driver to control a printer, only a low-level language will do.

SQL is covered in more depth in Chapter 8, beginning on page 314.

THINK ABOUT COMPUTING 6.4

Research three popular languages to discover their origins. What did existing languages lack that led to the need for the new languages?

C has both high-level features (for example, FOR loops and arrays) and low-level features (for example, memory pointers and byte-level operators) making it popular and suitable for many occasions.

Software development tools

To develop software, you need a number of basic, essential tools, including an **editor**, **compiler**, **linker** and **debugger**. The following section discusses these in more detail.

An editor is a specialised word processor that is used for creating human-readable **source code**, or rather, human-readable programming instructions. Code editors come with specialist features designed to make programming easier, such as highlighting programming keywords, detecting unbalanced parentheses and adding line numbering. Programming editors, such as the one shown in Figure 6.2, show colour-coding, indenting, collapsible text and line numbering.

A compiler converts source code into executable programs that a computer can carry out; that is, that a particular **central processing unit (CPU)** and operating system, such as Windows or Mac OSX, can understand.

```
321
322 function getState(  stsDOC, stsDSC, stsDJS, stsDBS, stsDWS )
345
346 function getStateReason( stsDWS, stsDOC, stsDSC )
347 {
348         var stateReason = '';
349
350         if( typeof stsDWS !== 'string' || stsDWS === '' ||
351             typeof stsDOC !== 'string' || stsDOC === '' ||
352         typeof stsDSC !== 'string' || stsDSC === ''
353         ) {
354                 return '';
355         }
356
357         if (stsDSC !== 'NO' || stsDOC !== 'NO') {
358             stateReason = 'AttentionRequired';
359         } else if (stsDSC === 'NO' && stsDOC === 'NO' && stsDWS === '1900') {
360             stateReason = 'Paused';
361         } else if (stsDSC === 'NO' && stsDOC === 'NO' && stsDWS === 'NO') {
362             stateReason = 'None';
363         } else {
364             stateReason = '';
365         }
366
367         return stateReason;
368 }
369
```

FIGURE 6.2 A programming editor

THINK ABOUT COMPUTING 6.5
Research online to choose a popular code editor. Examples are Notepad++ and Programmer's Notepad. What features does it possess that make it better than a plain text editor for creating source code?

Executable code compiled for one **platform** will not work on another without being ported (re-compiled for another platform). Porting is hard and expensive work; this is one reason why many apps are available for Windows but not Mac or vice versa.

Part of the difficulty of porting is that programs must often be substantially changed to work under a different operating system.

A linker loads information that the executable code will need; for example, how to read a keyboard or how to calculate square roots. Linkers come with useful pre-written code libraries.

A debugger helps programmers to find bugs – or programming errors. Sometimes debugging can take as long as the original programming time, or even longer if the program has been poorly designed. Debuggers may:

- highlight incorrect **syntax** (programming expression) and show how statements should be expressed
- allow programmers to set **break points** in code, where the compiler will stop and let the programmer inspect the current values of variables
- allow line-by-line **stepping** through code so developers can find exactly where a problem arises.

Few programmers today use separate editors, compilers, linkers and debuggers. Most use an **Integrated Development Environment (IDE)** such as Microsoft's Visual Studio to combine the development tools into a single package.

Figure 6.3 shows:

1. the **toolbox** of **graphical user interface (GUI)** objects the programmer can insert into the program
2. the **code window,** which the programmer uses to instruct the program how to act when an event takes place, such as the clicking of a button
3. the **form**, which will be the visible **interface** for the program's user
4. the **properties** of the selected **object**, which let the programmer modify an object's characteristics or behaviour
5. the **project manager** with which the programmer can manage the various files and components related to the program
6. **onscreen help** to give brief reminders of what the currently selected object is like.

Refer to page 253 for more information on GUI objects.

Refer to page 235 for more information on interfaces.

Refer to page 232 for more information on objects.

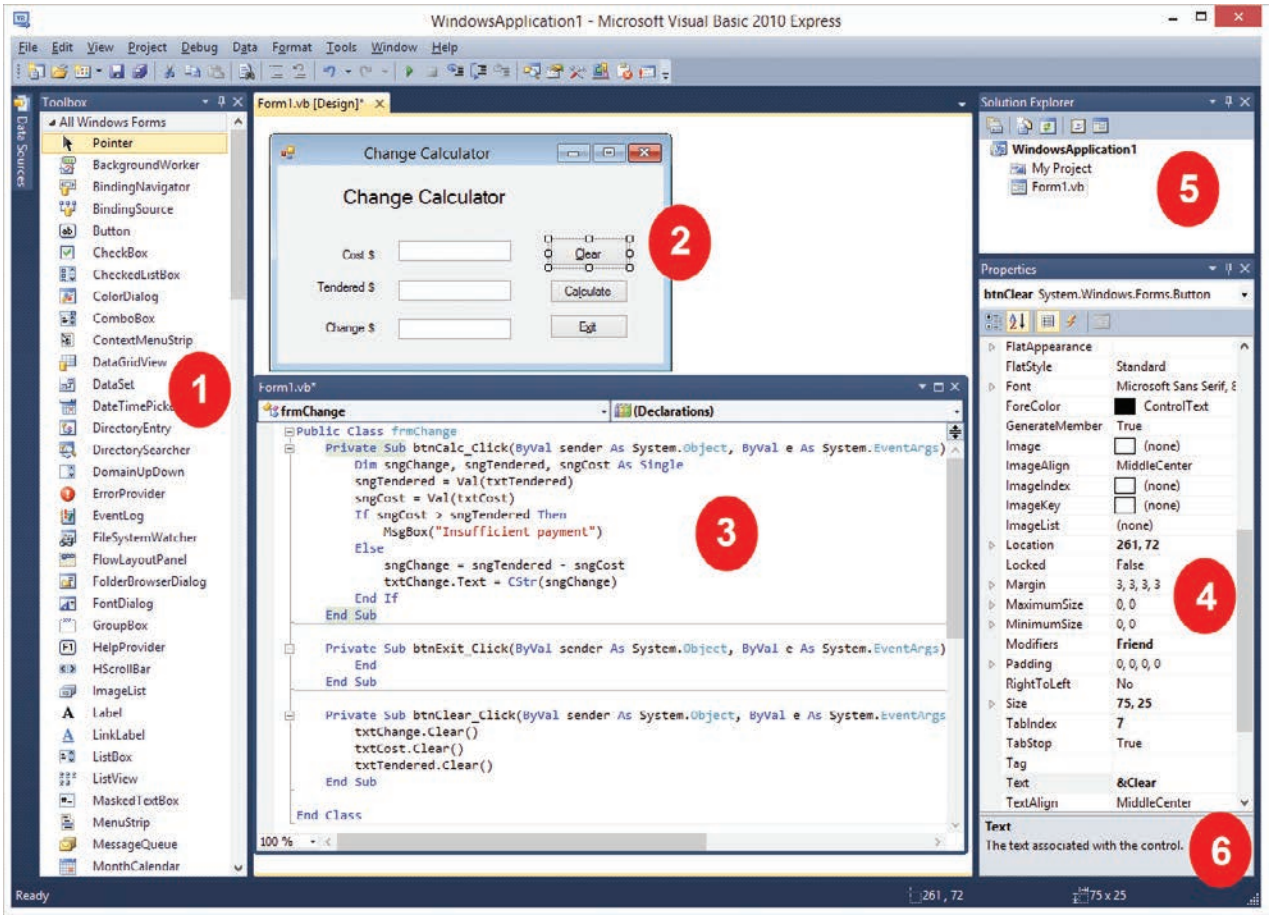


FIGURE 6.3 An example of an IDE

Storage structures

A storage structure is a location in RAM where data is stored during the execution of a program. The two main data storage structures are **variables** and **constants**. Variables are so-named because the value stored in a variable can *vary*, or be changed by a program. Constants, on the other hand, have fixed (unchanging) values during a program's execution, such as the value of pi, or the number of the Australian states and territories.

Arrays can store many values in numbered 'slots'. For example, to store 12 monthly rainfall figures, create an array called `intRain[12]` to house 12 values. To address (refer to) an individual value, give the name of the array and the desired **index** (slot number); for example:

```
IF intRain[1] < intRain[12] THEN
  DISPLAY "January was drier than December!"
END IF
```

Using variables and arrays is discussed later in this chapter.

Data types

Most languages want programmers to declare the *type* of data that needs to be stored, so they can most efficiently store the data. After all, if someone asks you for a box, do they want to store a pair of shoes, a wild cat or a widescreen TV? Knowing the intended contents lets you choose the container of the best size and type.

Chapters 7 and 8 discuss data types in greater depth.

All programming languages support various data types for variables and arrays, but they differ in what types they support. Most languages support text (string), integers, floating point (real) numbers, date/time (timestamp), and Boolean data types, as described in Table 6.2.

TABLE 6.2 Common data types

Data type	Description
Text (string)	Alphanumeric characters and punctuation – any group of characters that can be typed, such as `Tom Smith, *Hello*` or `123abc`. Numbers can be stored as string, but cannot be used for calculations in that form. Use strings to store phone numbers like `(03) 3945 2394`, because strings can hold parentheses, leading zeroes and spaces while numeric types cannot.
Integers	Whole numbers with no fractional part, such as `1, 0, 3, 67` and `341567` Fractional data is lost when stored as integers. Integers require little RAM.
Floating point (real) numbers	Numbers with fractional parts, such as `2.42`. Floating point numbers use more RAM than integers. Types include single precision and double precision.
Date/time (timestamp)	Calendar date and/or time of day. Using date/time data type allows programming languages to perform complex time and date calculations.
Boolean	Stores only two values: true or false. Although this may not sound very useful, computers spend a lot of time making true/false decisions and the dedicated, concise Boolean data type saves a lot of RAM.
Character	A single character only, such as `M`, `F` or `$`
Byte	An integer between 0 and 255
Currency	Numeric, for storing monetary values only; it has a large number of decimal places to prevent rounding errors
Pointer	C uses this special data type to store pointers to the locations of items in RAM

Languages differ in how they treat floating point numbers. A single precision variable may use 4 bytes of memory to store a value up to about 38 digits long, with 7 decimal places. Double precision is much bigger!

When you select a data type, you should choose wisely so that you will not lose vital information. For example, if you chose to store a floating point number in an integer variable, it would lose its decimal places because, as described in Table 6.2 above, integer data types do not store decimal places. Conversely, small integers do not always need to be stored as floating point numbers, particularly very large ones such as double precision, because they are *small* and using large data types will just waste RAM and slow down your execution. Thus, when choosing data types, consider the form of data and its size.

THINK ABOUT COMPUTING 6.6
Look into the specifications of the programming or scripting language you are using and find the minimum and maximum values that it can store in an integer.

You should also think about both current and future needs. For example, Kelly is programming an employee database for a company that has 30 employees. She chooses 'byte' data type for the variable to hold the number of employees. Six years later, the company has grown considerably and now has 256 employees. Kelly's program crashes as soon as employee number 256 is entered into the system, because byte variables can only store numbers up to 255. Kelly's conservative choice of data type killed her software.

THINK ABOUT COMPUTING 6.7
List the data types your language supports. Identify their purposes.

Media data types

A **medium** is a channel or means through which data and information are sent or stored. In information and digital systems, multimedia means the use of several media, such as video, audio, text and photographs, to convey information. Storage media are ways of storing data, and include Blu-ray, SSDs, HDDs and the cloud.

Media are data-hungry. An 30-minute audio file uses 6–30 MB of storage. Video can consume 20 times that amount. A digital photo can be anywhere from a few kilobytes (KB) to several megabytes depending on its quality.

Byte measurements are confusing. Since 1998, official IEC standards say that a kilobyte is 1000 bytes and 1024 bytes is now a 'kibibyte'. Many ignore that and use the traditional 1024 byte kilobyte.

Data storage is measured in bytes. One byte is made up of 8 binary bits (1 or 0). A kilobyte (KB) is approximately 1000 bytes. A megabyte (MB) is approximately 1 million bytes. Transmission speeds are usually measured in bits per second, not bytes. In abbreviations, byte is represented by the uppercase 'B', and bits use 'b', so 56 Mbps means 56 million bits per second (7 million bytes per second). Do not confuse bits and bytes! Refer back to Table 6.1 on page 221 for more information on data storage units.

The quality of stored media involves their resolution and compression, which is discussed in the next section.

Resolution

Resolution is a measure of how much detail is in:

- images (dots per inch when printed or pixels per inch when onscreen and colour depth)
- video (frames per second, frame size and bitrate)
- audio (sample rate or frequency, bit depth and number of channels).

The higher the resolution, the more realistic the reproduction will be, and the bigger the media file… and the longer it will take to transmit. Choosing a level of resolution for media usually involves a compromise between quality, cost, storage requirements and the time needed to transmit.

THINK ABOUT COMPUTING 6.8
Research the resolutions used for laser printing, phones and monitor screens. How do the differences affect file size?

When saving JPEG files, the 'Quality' compression number is not a percentage. It just means 'a little bit' or 'a lot'.

Compression

Bulky media are usually compressed before being stored or transmitted.

Lossy compression shrinks media by throwing away details. Common lossy media formats are used to shrink:

- photos, such as JPEG
- video, such as DivX, MKV and WMV
- audio, such as MP3 and WMA.

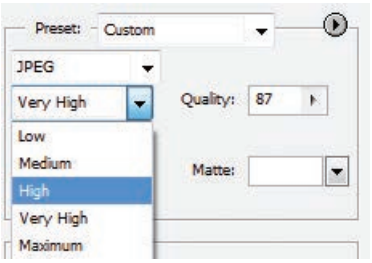


FIGURE 6.4 Setting the compression level when using the 'Save for Web' option and saving as a JPEG in Photoshop.

JPEG collapses similar colours into a single shade. MKV may reduce the number of frames per second. MP3 removes low or high musical notes that most people cannot hear. When users choose a level of lossy compression, they must balance quality against size.

THINK ABOUT COMPUTING 6.9
Using free software such as Audacity take an audio file and save it with different sample rates (e.g. 8 KHz and 44.1 KHz) and sample sizes (e.g. 16-bit or 32-bit) in mono and stereo. Compare the sizes of the files and sound qualities.

Detail lost during lossy compression can *never* be restored. Always keep original, uncompressed master copies of media for later use.

Lossless compression file formats (such as GIF, PNG, TIFF images, and FLAC audio) reduce media size as much as possible without losing data. They work by summarising data. For example, instead of recording 500 blue pixels individually, it says, 'The next 500 pixels are blue'.

JPEG format is designed to store photos with subtle colour transitions. GIF is designed to store images with big blocks of solid colour, using a palette of only 256 colours. This explains why JPG is a poor choice for storing logos, as GIF is for storing photos.

Text is never stored in a lossy format; throwing away little words or changing big words into similar smaller words to save space would not be not wise! Text formats include:

- plain text (txt), which stores nothing but text characters
- comma-separated values (CSV) or tab-separated data files
- rich-text format (RTF), which includes formatting tags, such as HTML
- portable document format (PDF), which is a searchable, compressed format that can be viewed in its original and perfect formatting without using the obscure or expensive software that created it
- proprietary formats such as DOCX and WPD that have detailed formatting, images and metadata (data about the text).

Image colour information is stored in two different ways. Images to be viewed on monitors are stored as RGB with three bytes representing the brightness of a **r**ed, a **g**reen and a **b**lue pixel between 0 (off) and 255 (fully on). Images intended for printing use CMYK to describe the strengths of the four ink colours – **c**yan, **m**agenta, **y**ellow and blac**k**.

THINK ABOUT COMPUTING 6.10
Find a digital logo with a couple of solid colours, and a digital photo of a landscape. Using an image editor like Photoshop or GIMP, load both images and save them under both JPEG and GIF with new names. Compare the file sizes and image qualities. Now try saving the JPEG image with increasing compression levels. Graph the sizes and subjective assessments of the image quality.

FIGURE 6.5 A document in print layout when using word processing software

```
{\plain \fs36 \b\f1\fs36 {\tc {This is a main heading}{\plain \
fs36 \b\fs36 .}}\par
}\pard \fs24
{\plain \fs24 \b This text is in }{\plain \fs32 \b\fs32 bold}{\
plain \fs24 \b .}{\plain \fs24 \par
}{\plain \fs24 \i Italics here.}{\plain \fs20 \fs20 \par
```

FIGURE 6.6 Part of the same document, saved as RTF

Problem-solving methodology is explained further in Chapter 2.

Developing software

The software development used in VCE Computing can be applied either as the single stage-by-stage process of the problem-solving methodology (PSM) or to each iteration of an agile problem-solving process. An agile process is a flexible and responsive approach that allows a return to earlier production steps when needs change; for example, if changes in technology force a radical redesign of the product, the client wants to add new functionality, or market pressures force a change of direction.

PSM stage: Analysis

The task of this stage is to develop the software requirements, both functional and non-functional.

Functional requirements

Functional requirements describe *the tasks* that a program should be able to perform. In the simplest terms, these are the things a program must be able to do – the main reason for creating it. For example, a program's functional requirements may specify that it must be able to:

- edit, crop and touch-up illustrations and photos
- create illustrations
- create handwriting typefaces.

The functional requirement or requirements are usually achieved in a specific and identifiable place in a program or a solution, such as a particular formula or a piece of programming.

Non-functional requirements

Non-functional requirements describe the *attributes* or *qualities* that your solution should have. Using the design program as an example again, its non-functional requirements may require it to be the following: precise, flexible, fast and easy to use.

A non-functional requirement will probably not be achieved in one specific place in a program. It usually requires a combination of factors across an entire program. For example, achieving ease of use in a design program may involve building in simple menus, shortcut keys, context-sensitive help, using clear language and making sure the interfaces are user-friendly in many different places.

The results of the analysis stage, comprising solution requirements, constraints and scope, are recorded in a document called solution requirement specifications, or SRS. In this Outcome, your teacher will give you the analysis.

Constraints

Constraints are limiting factors or conditions that you need to consider when you are designing a program. A constraint will usually reduce your freedom of design choice. Constraints generally fall into five categories: economic, technical, social, legal and useability.

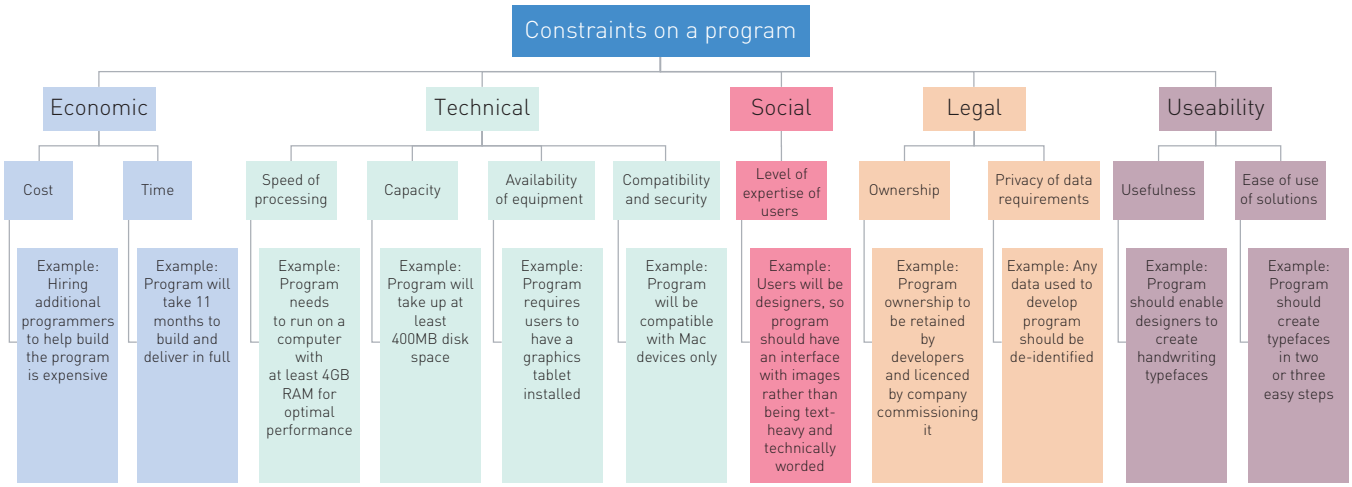


FIGURE 6.7 Constraints on a program

PSM stage: Design

Design is a vital stage in starting to create a good program, and there are several acknowledged good practices to observe.

Software design uses various tools to plan a program's architecture (how it will be constructed) and its appearance. The following sections discuss several types of software tools that can be used to help plan a program's architecture: data dictionaries, data structure diagrams, input-process-output charts, pseudocode, object description and interface mock-ups. You may find these tools useful during the development of your Outcome. Each design tool has its own specific purpose and reason for existing. If you ever find yourself using a design tool that gives no more information than that already provided by other design tools, you are probably using it incorrectly.

TABLE 6.3 Tools to plan a program's architecture and appearance

Design tool	Designs
Data dictionary	Data types, names, formatting, validation
Input-process-output (IPO) chart	Output and data requirements, calculation strategies (algorithms)
Data structure diagram	Structure and relationships within and between data items
Object description	The behaviour and properties of components
Pseudocode	The logic behind processing
Mock-up	The appearance of output and the user interface

Data dictionaries

A data dictionary is used to plan storage structures including variables, arrays, and GUI objects such as textboxes and radio buttons. The data dictionary should list every structure's name and data type. It may also include the data's purpose, source, size, description, formatting, and validation.

For more information on data dictionaries, refer to Chapter 8, page 312.

TABLE 6.4 Data dictionary

Name	Type	Format	Size	Purpose	Example
`txtCustomerID`	Text	XXX99	5	Customer ID	SMO40
`dateDOB`	Date	YYYY-MM-DD	Fixed	Date of birth	1992-12-28
`sngSales`	Single precision	$##,###.##	Fixed	Total amount spent	$12,456.78
`boolClubMember`	Boolean	Yes/No	Fixed	Is a member of the buyer's club?	Yes
`txtFamilyName`	Text	Xxxxxxxxxxx	25	Customer family name	De Silva
`txtFirstName`	Text	Xxxxxxxxxxx	15	Customer given name	Horatio
`intAge`	Integer	999	Fixed	Age in years	34
`intMemYears`	Integer	99	Fixed	Years a member	12

Data dictionaries are valuable when code needs to be modified later by other programmers and the purpose of a variable or array is unclear.

Object naming

Creating clear and obvious names for your program's variables, arrays, GUI controls, forms and windows makes your source code more readable and maintainable over time. It also makes it more easily understood by programming colleagues, and will save you some time after you have finished the code.

Descriptive names make it easier to know what an object is for, and how it should be treated. For example, the name 'Temperature' is much more informative than 'T'.

There are two industry-standard naming techniques.

Hungarian notation involves adding an object's type as a prefix to its name, such as `intTemperature` for an integer, or `lblHeading` for a label. This practice is particularly common in databases and programming. The prefix reminds programmers how the object should be handled; for example, you would not accidentally try to change the Multiline property of `lblHeading` because the `lbl` party reminds you that it is not a textbox and does not have a Multiline property.

CamelCase involves the use of capital letters to mark the start of new words in a file or object name. Multi-word names can be hard to read in a name that has no spaces, such as `inttemperaturecelsius`, but programming languages forbid spaces in names. In situations like this, CamelCase helps by capitalising the initial letter of each word in a name, so `inttemperaturecelsius` becomes `intTemperatureCelsius`. Using another example, in the filename `annualcompanyauditreport2017.docx`, the initial capitals system of CamelCase would make things much easier: `AnnualCompanyAuditReport2017.docx`.

The way that files are named is very important. Get started with best practices early.

- Use underscores (underlining) instead of spaces; for example, `z_dev_MediaGrid.fmp` instead of `z dev MediaGrid.fmp`.
- Keep names short but meaningful; for example, some of the most common programs have short but meaningful names: WINWORD.EXE, Acrobat.exe, and so on.
- Be consistent; for example:

  - intTemperatureCelsius.xlsx, intTemperatureFahrenheit.xlsx, intTemperatureKelvin.xlsx
  - inttemperaturecelsius.xlsx, intTemperatureFahrenheit.xlsx, int Temperature Kelvin.xlsx

- You will be unable to use characters forbidden by the operating system, which include the following punctuation marks in Windows:
  * ? < > : / \ " |

Data structure diagrams

A data structure diagram shows the structure and relationships within and between the data items in the data dictionary. A data structure diagram does not repeat information already present in the data dictionary; that would be a waste of time.

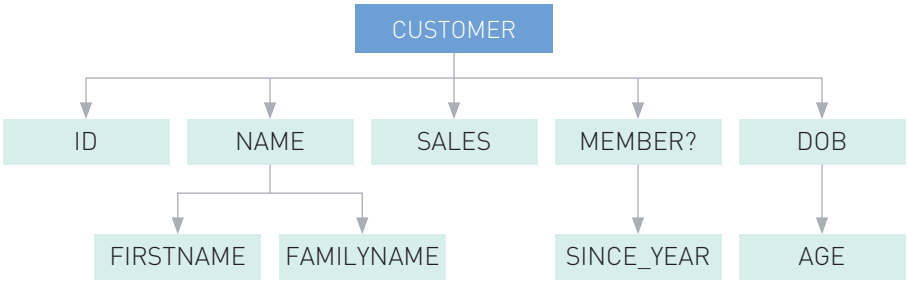


FIGURE 6.8 Example of a data structure diagram

Input–process–output (IPO) charts

IPO charts help programmers to design formulas and **algorithms**. An algorithm is the strategy for a calculation. A few steps must be followed to create and complete a chart correctly. First, enter the information required, such as a person's age, into the **Output** column.

Next, ask what data is needed to calculate that output. To calculate a person's age, we need to know two things: Date of birth (DOB) and the current date. Enter these into the **Input** column.

Finally, work out what kind of processing (algorithm) needs to be done on the input to calculate the desired output. Enter this algorithm into the **Process** column as pseudocode – do not write source code.

The second row of Table 6.5 below uses our example of calculating a person's age in years. The algorithm in the Process column describes a technique for calculating the answer. Find and then subtract the number of days between the birth date and now, and divide that by 365 to get years. The IPO chart can now be completed.

Row three of Table 6.5 calculates a subtotal by multiplying quantity by cost per item. Row four calculates total cost by taking a subtotal amount and adding tax (if payable) by multiplying the tax rate percentage by the subtotal. Row five uses a condition: if age is greater than 60 years, then total cost should be total cost minus the total cost divided by discount rate percentage. This becomes the senior citizen cost, the output.

A bad algorithm can make a program slow, fat or unreliable. A good algorithm can make a program responsive, small and trustworthy.

Every calculation – even the really simple ones – should be designed and put in the IPO chart. Later programmers might need that information.

TABLE 6.5 An IPO chart

Input (data)	Process (algorithm)	Output (information)
DOB Current date	(Current date – DOB) / 365	Age in years
Quantity Cost per item	Quantity * Cost per item	Subtotal
Is tax payable? Tax rate %	Subtotal + (If tax is payable, subtotal * tax rate %)	Total cost
Age in years Total cost Discount rate %	If age in years >= 60, Total cost – (total cost * discount rate %)	Senior citizen cost

See Chapter 2 for more information on IPO charts.

Notice how the information from previous calculations is often used in later calculations.

The IPO chart can then be given to a programmer and the algorithm converted into source code for any chosen programming language.

Object description

In programming, an object is any item that a program can inspect and/or change in terms of its appearance, behaviour or data. Today, object-oriented programming (OOP) is a common practice. OOP focuses on objects, such as icons, menus, buttons and listboxes, in a GUI that a user manipulates to issue commands and display information. OOP objects have properties, **methods** and events.

1 Properties are the attributes of an object, such as width, colour, size, name and visibility. For example:
`listbox1.` **`width`** `= 200` sets a property.
2 Methods are the actions that an object can carry out, such as move, refresh, setfocus or hide. For example:
`mainwindow.` **`refresh`** uses a method of the window.
3 Events are actions or occurrences that an object can detect and respond to accordingly, such as a mouse click, key press or a timer going off. Each event usually has its own procedure, which describes what will happen when the event occurs. For example:
`txtFamilyName.` **`keypress`** responds to a user's typing into the `FamilyName` textbox object.

An object description is a way of describing all of the relevant properties, methods and events of an object.

```
OBJECT: txtName
PROPERTIES

Class: textbox
Left position: 300
Width: 500
Font: Arial
Justification: left
Visible: yes
Font colour: black

METHODS
Cut: save cut text to disk

EVENTS
Keypress: if key is CTRL+[ set text justification to left.
```

FIGURE 6.9 Example of an object description

Pseudocode is probably the most important tool for students of Software Development VCE Units 3 & 4.

Pseudocode

Writing an algorithm in source code is slow. An algorithm written in source code also limits itself to use in only one compiler. **Pseudocode**, also known as Structured English, is a quick, flexible, and language-independent way of describing a calculation strategy – halfway between

English and source code. Once the algorithm is sketched out in pseudocode, it can be converted into source code for any desired programming language.

'Pseudocode' literally means 'false code'.

A good algorithm can be extremely valuable and bring forth great change. A clever strategy can make software run twice as quickly or use half the amount of RAM. An ingenious idea can lead to the development of a program that was once considered impossible. For example, Google's PageRank completely changed the way the world searched the internet, and made billions of dollars for its inventor in the process. The invention of **public key encryption** finally cracked the age-old problem of how to encode and transmit secrets without having to also send an unlocking key that could be intercepted.

Google PageRank checker

This pseudocode determines if a year is a leap year:

```
if (year is divisible by 4 and not divisible by 100)
or (year is divisible by 4 and 100 and 400) then
    it's a leap year
else
    it's not a leap year
end if
```

THINK ABOUT COMPUTING 6.11

Use a testing table to check whether the leap year algorithm works for the years 1999, 2000 (which *was* a leap year), 2020 (also a leap year) and the current year.

The rules of pseudocode

What are the rules of pseudocode? Easy: there are none. As long as the intention of the calculation is clear, it is good pseudocode. If not, it is bad.

However, ensure that you specify assignment (the storage of a value) using the ↗● symbol rather than the equals sign (=) that is used in algebra and in most real programming languages; for example:

```
IsLeapYear ↗● True
```

The equals sign is reserved for logical comparisons, such as:

```
IF B=0 THEN CALL SoundAlarm
```

Common features found in pseudocode include:

- loops, such as `WHILE/ENDWHILE` and `FOR/NEXT`
- control structures, especially `IF/ELSE/ENDIF` blocks
- logical operators – `AND`, `OR`, `NOT`, `TRUE` and `FALSE`
- arrays, such as `Expenses[31]`
- arithmetic operators `(+ - * /)` and the familiar order of operations, as used in Year 7 Mathematics and Microsoft Excel spreadsheet formulas.

Pseudocode punctuation and the names of key words are largely up to you if it is clear what you mean; for example, it does not really matter if you prefer `WHILE/WEND` or `WHILE/ENDWHILE`.

To 'Get data from keyboard', you could use `INPUT`, `GET`, `FETCH`, or another keyword. To read data from a disk file, you could choose `INPUT`, `GET`, `READ` or something else. To avoid ambiguity, you could explain your pseudocode's conventions using comments, as shown in the example on page 234.

```
# The hash symbol precedes a comment
# GET reads the keyboard.
# READ loads data from a disk file.
# DISPLAY shows output on screen.
# WRITE saves output to a file.
DISPLAY "What is your name?"
GET UserName
OPEN FILE "Users.txt"
READ data for UserName
IF new data exists THEN
    WRITE new data to file
END IF
```

Interface mock-up

If software will be used directly by people (rather than running hidden deep in the OS), it needs an **interface** – a place where people can control the program, enter data and receive output. A successful interface cannot be cobbled together. It must be carefully designed to make it usable and clear.

To design an interface, use a **mock-up**, which is a sketch showing how a screen or printout will look. A mock-up should typically include the following features.

- The position and sizes of controls such as buttons and scroll bars
- The positions, sizes, colours and styles of text such as headings and labels
- Menus, status bars and scrollbars
- Borders, frames, lines, shapes, images, decoration and colour schemes
- Vertical and horizontal object alignments
- The contents of headers and footers

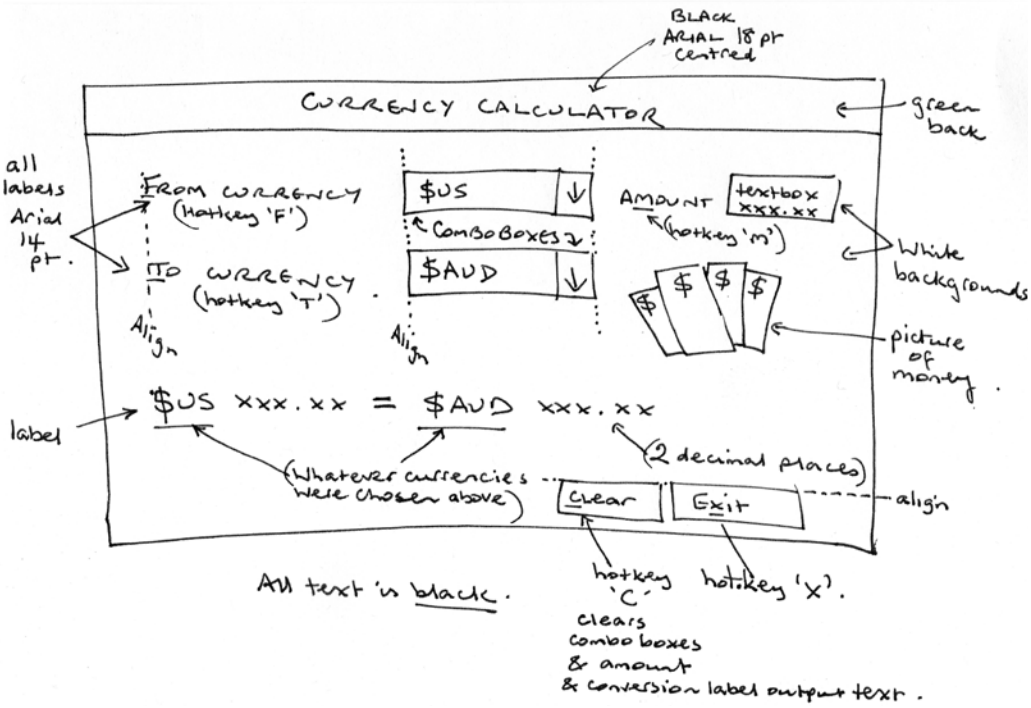


FIGURE 6.10 A mock-up of a screen interface

In VCE Computing it is not mandated that you use software to create your mock-ups. You may use software such as Balsamiq Mockups if you wish, but you may also create them by hand using pen and paper.

A mock-up can be considered successful if you can give it to another person and they can create the interface without needing to ask you questions.

Creating effective user interfaces

Very good interfaces are difficult to make because human beings are individuals. We have our own preferences and operate differently depending on gender, experience and cultural background. However, to create usable interfaces, the same several factors must be always considered and applied: useability, accessibility, structure, visibility, legibility, consistency, tolerance and affordance.

Useability

Software needs to do more than just create accurate output. It must let users work efficiently and require minimal learning, memorisation and stress. You should make commonly used features the quickest and easiest to find. Do not hide basic functions deep in a menu, because users will find it frustrating.

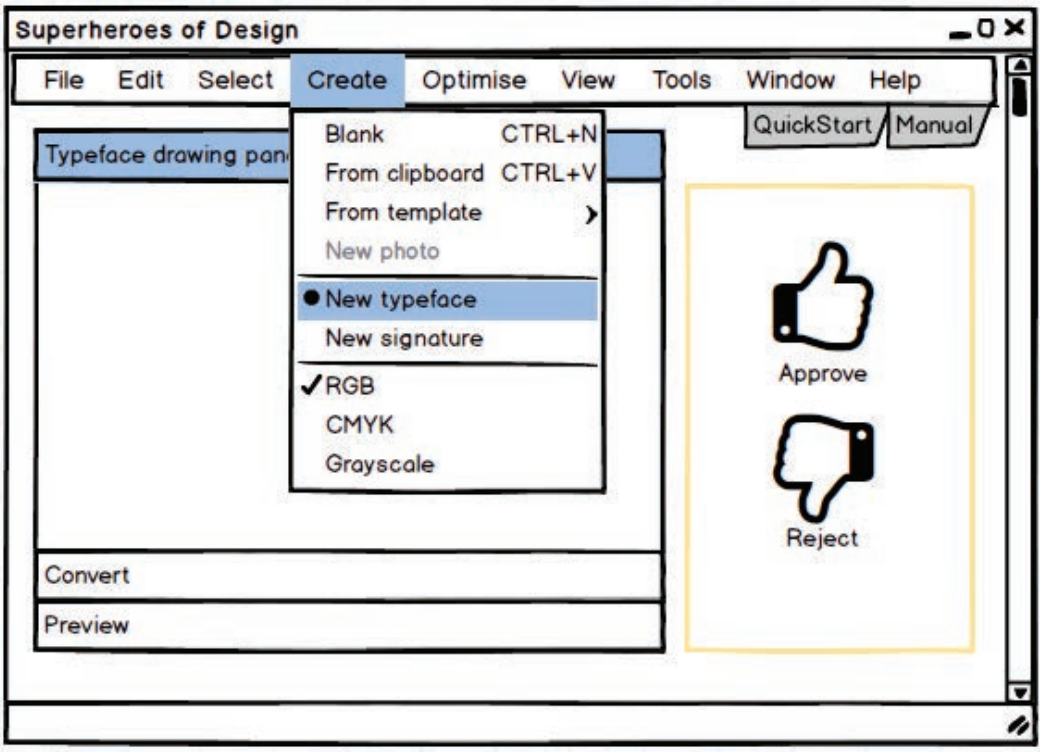


FIGURE 6.12 Ease of use: Although this mock-up looks easy to use, users would normally expect that creating a new document of any kind, such as the new typeface, is usually a menu option contained within the 'File' menu at the far left, which may make this interface *less* useable. This is something to consider.

There are many ways to provide help for your users. Consider providing printed manuals, quick start guides, internet help guides, context-sensitive help, onscreen instructions, pop-up tool tips and/or examples.

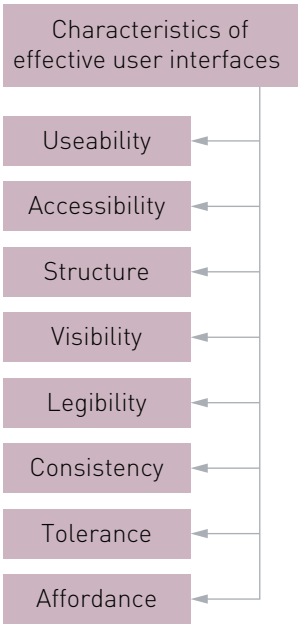


FIGURE 6.11 Characteristics of effective user interfaces

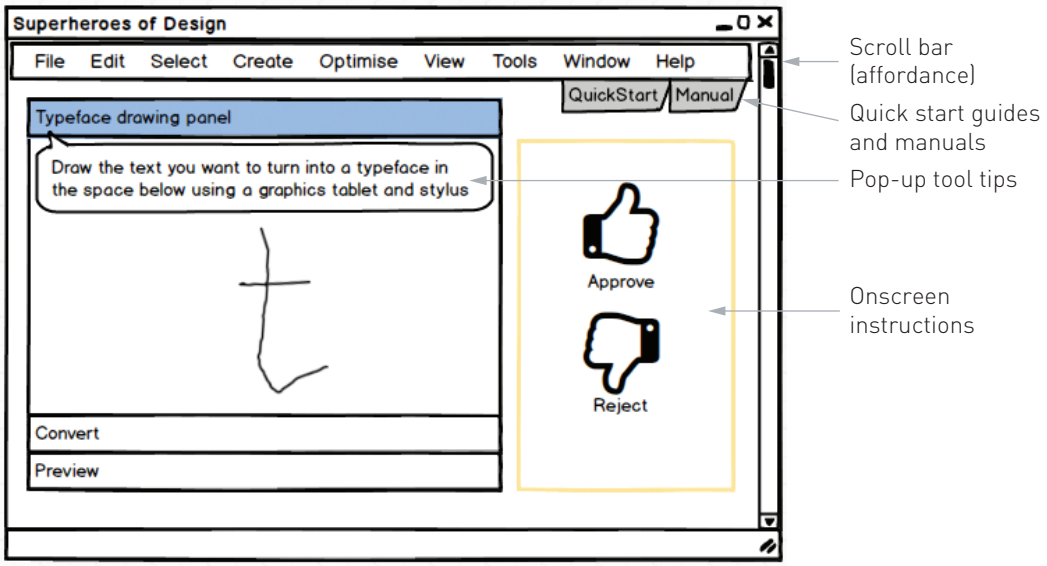


FIGURE 6.13 Tool tips, quick start guides, manuals and onscreen instructions make this interface somewhat easy to use. What else would make this interface even easier to use?

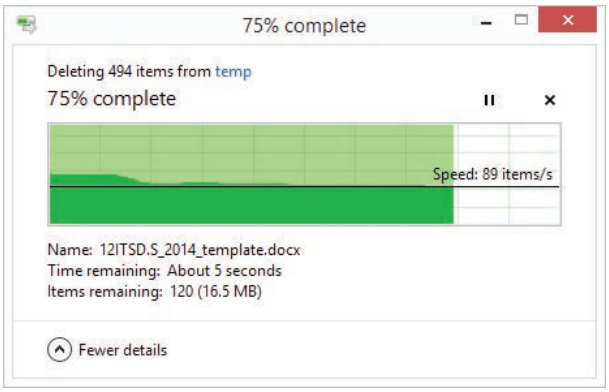


FIGURE 6.14 Showing progress during a long operation

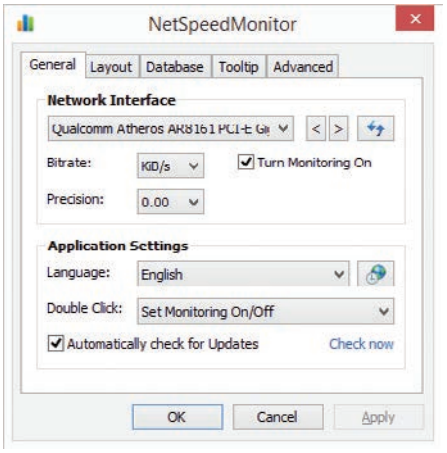


FIGURE 6.15 A small form using tabs and dropdown lists to prevent overcrowding

THINK ABOUT COMPUTING 6.12
What forms of colour blindness exist?

Another useability factor to consider is showing progress. Few things cause users more anxiety than when a program gives no indication about what it is doing. Has it frozen? Is it deleting everything on the computer? I need to leave in five minutes – what is this machine doing? Computer users panic easily when programs stop communicating with them. Give the user some reassuring feedback about the operation's status with a progress bar, an estimate of time remaining, a spinner – anything to stop users reaching for the unresponsive computer's reset button (Figure 6.14).

Affordance

Affordance refers to the concept that objects on your interface should immediately suggest what they do and how to use them. An interface with good affordance naturally leads people to use it accurately, efficiently and intuitively to accomplish their goals.

- The shadow effect on a button suggests it should be clicked with a mouse.
- A scroll bar looks like the natural thing to do would be to drag it up and down.
- A flashing red icon instinctively suggests a problem.

Accessibility

In software development, **accessibility** refers to catering for the disabilities or special needs that your software's users may have.

- To cater for colour blindness, avoid putting green text on a red background.
- To cater for poor eyesight or low vision, do not make text too small.
- To cater for those who have limited hand coordination, make buttons larger and space them further apart.
- To cater for those who have limited reading ability, use short words and avoid colloquialisms and jargon.

Structure

You should organise any user interface that you design quite deliberately so that it makes sense to end users. You could base it on existing models that users are already familiar with, rather than changing fundamentals. For example, Figure 6.12 moves the creation of a new typeface document from the 'File' menu to a new menu 'Create' away from the top left, and this is a needless shift. It differs from a working model that is what users already expect. Part of a working structure that you could use would move this grouping back to the 'File' menu, as shown in Figure 6.16.

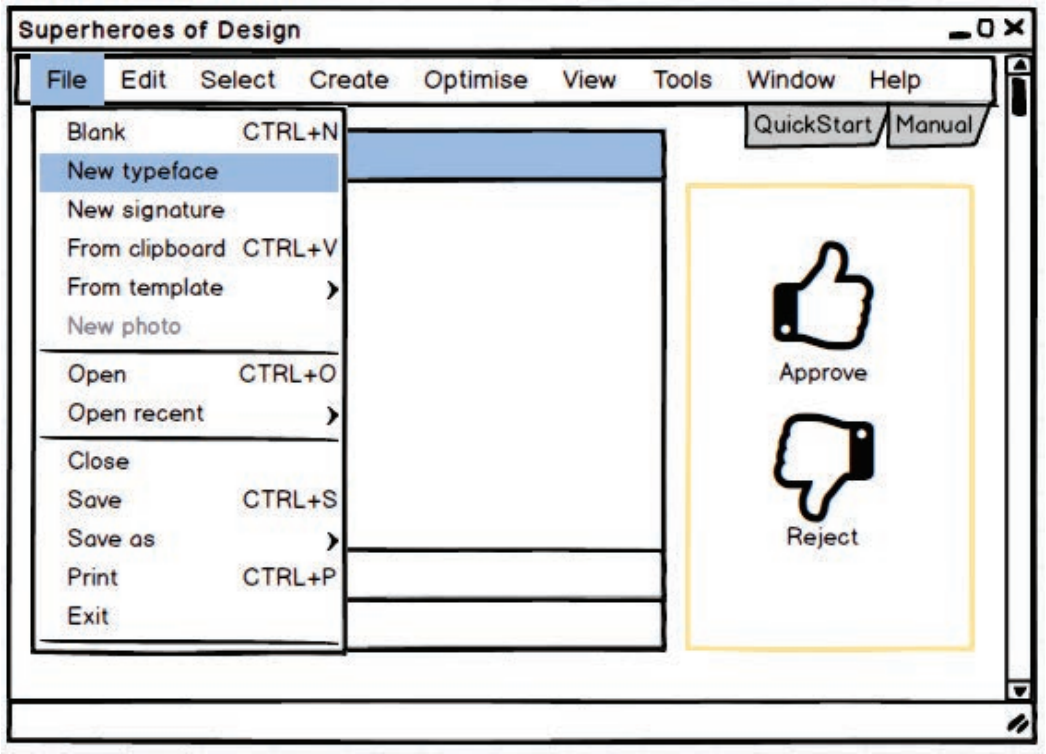


FIGURE 6.16 Structure based on a consistent, recognisable model

You should also put related items together so they are easier to find, and separate distinct items in clear, sensible groupings. Ensure that users do not need to hunt across several forms, menus or screens to carry out related actions. See the example of a GIMP interface shown in Figure 6.18.

Software users do not want to have to learn each programmer's personal stylistic preferences. They want to start a program and use standard techniques, knowing instinctively where things are and how they work. For example, in Windows, the software version is found under the Help menu > About, and the Help menu is always the right-most menu item. Do not put it elsewhere just to be different!

Software consistency is no accident. All major software developers publish **style guides** giving programmers instructions about how to design software for their platforms. To ensure software is predictable and easy to learn, some major software companies will not certify software unless it obeys their guidelines.

THINK ABOUT COMPUTING 6.13
Download interface design style guides from Microsoft and Apple (search for 'windows style guide' or 'apple style guide'. How do their styles differ?

Visibility

Visibility in a user interface means that the tools and options that the user needs to perform a specific task should be visible to them without them being distracted by superfluous information. If a user does not need to see extra information to make a decision to approve or reject a character

they have drawn in their typeface, then do not show it to them. For example, the thumbs up or thumbs down Approve/Reject step shown in Figures 6.12, 6.13 and 6.16 will suffice.

You should only show as much information as a user needs to make a decision or proceed to the next step, and no more. Overwhelming visual detail can make an interface confusing and undesirable.

Instead of squeezing many objects onto a small form, use multiple forms, tabbed controls, pull-down menus and combo boxes that collapse when not needed (Figure 6.15).

An uncrowded interface is even more important when programming for mobile devices with small touchscreens that will be operated with big fingers.

Legibility

Promote ease of use and reading comprehension by ensuring that the information on the user interface is noticeable and clearly distinguishable, and that any text that appears on the interface is readable. (Refer to Chapter 2 for more information on design principles and formats and conventions, which cover the contrast aspect of legibility in more detail.) Legibility deals with aspects including contrast, leading, kerning, line length and font size.

For interfaces:

- avoid using text in all uppercase
- use underlined text only to indicate hyperlinks
- use familiar, plain typefaces designed for reading onscreen, because decorative typefaces can be hard to read
- beware of overusing bold, italic and other font effects – not everything needs to be called out
- left-align text
- apply a hierarchy, with important things larger than less important things
- use appropriate contrast between text colour and background colour to maximise readability
- make smart use of white space; an interface crowded with controls is ugly and can lead to errors if users accidentally click the wrong item.

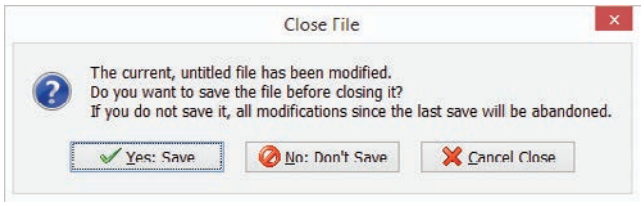
Tolerance

Tolerance is the capacity of software and interfaces to compensate for a user's errors and cope with people's natural differences in how they carry out tasks. An interface that forces users to obey its rigid expectations and is unforgiving of individual variation will be unpleasant and difficult to use.

Here are some examples of tolerant practice.

- Allow users to cancel or undo actions, and do not lock them onto a path from which they have no escape, such as printing 300 pages with no option to cancel.
- Provide settings and preferences so users can adjust a program's behaviour.
- Use the `MouseButton_Up` event rather than `MouseButton_Down` to trigger a button click. If users click a wrong button, a tolerant interface allows them to slide the mouse pointer off the button and safely let it go, while an intolerant interface would trigger the mouse event immediately, permitting no escape after the click.
- Warn users when they are about to do something dangerous or costly, such as deleting an account.
- Compensate for users' poor choices, such as by backing up data before deleting it – the Recycle Bin utility in an OS does this.
- Anticipate common errors and handle them gracefully. For example, Microsoft Word's Autocorrect quietly fixes typos like 'ACcidentally' caused by holding the SHIFT key down for too long at the beginning of a word.

- Make the **default action** the least harmful one possible. **Default values** should be the most popular and commonly chosen values. Courteous software lets users change default values.
- Make 'Cancel' the default button on a form so that no harm is done if the user carelessly hits ENTER.
- Provide users with choice. People use computers and software in different ways, and good software lets them choose how they do things, such as printing a document from the menu bar, from shortcut keys on their keyboard (such as CTRL+P in Windows), or from a button on a quick-access toolbar.



In Figure 6.17, by default, simply hitting ENTER triggers the 'Yes: Save' button rather than closing the file and losing data. The default button can be identified by its thicker border line.

FIGURE 6.17 The default button

Consistency

An interface should look and behave consistently from start to finish. Consistency should be applied in as many aspects as realistically possible. Refer to the GIMP interface shown in Figure 6.18 for an example of consistency. This includes:

- icons
- body text and heading styles
- text and background colours
- margins, borders, headers and footers
- navigation and menus.

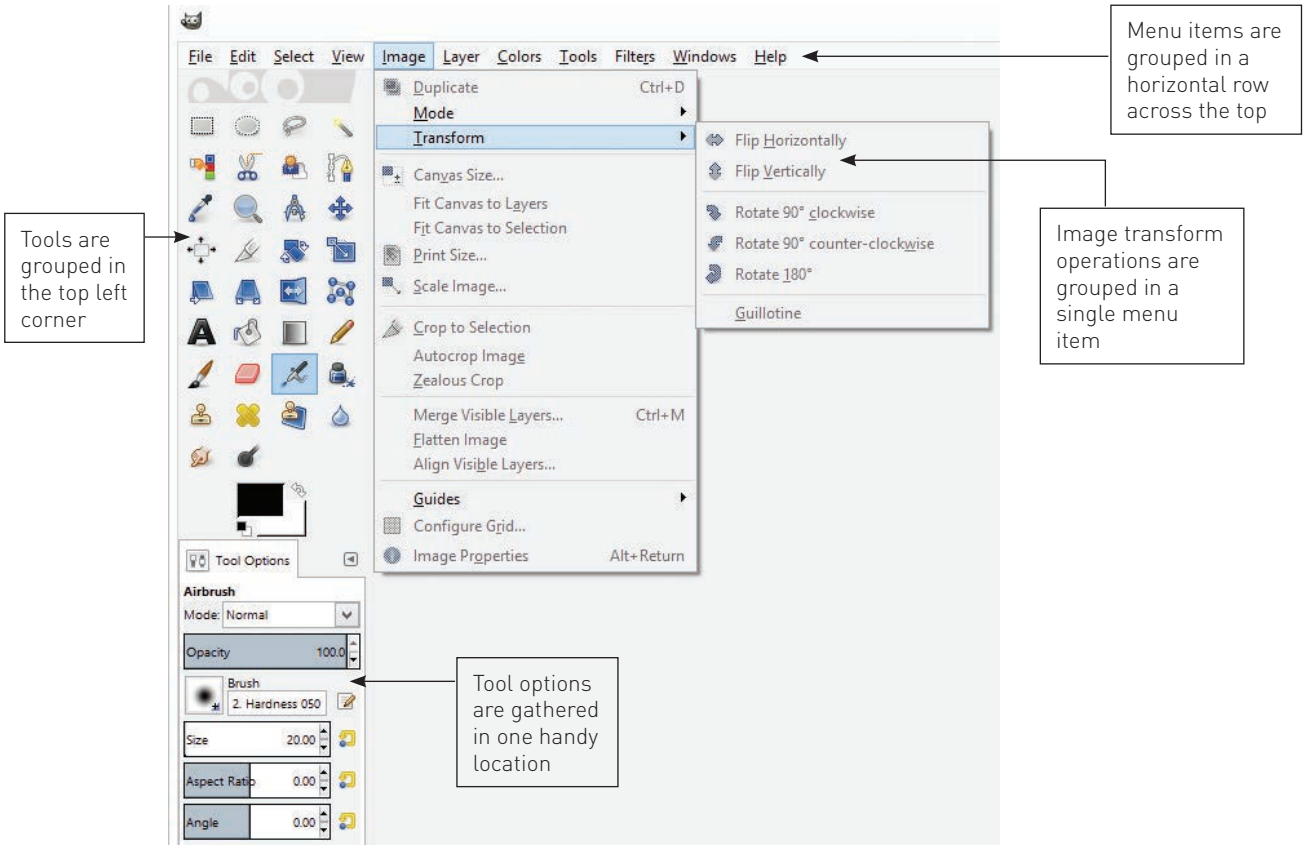


FIGURE 6.18 GIMP interface

Fundamental programming concepts

Despite the differences between platforms and languages, many concepts are universal to all programming environments.

Compiled and interpreted languages

Many programming languages, such as the C family, are compiled, meaning that source code is converted once by a compiler into executable code, such as an EXE file in Windows, to be run many times under a particular OS.

Scripting and interpreted languages, such as Python, PHP, Perl and JavaScript use a different approach. The source code is compiled *every time* it is run, instead of being compiled once by a compiler into a stand-alone executable program. This process is slower than once-off compiling, but:

- programs can be easily, swiftly and repeatedly modified by the programmer or end user without a compiler
- the code is human-readable and no viruses can be hidden in the source code
- source code only needs to be written once for all computers on all platforms that have the interpreter to run it; this is important for server-side internet programming.

All OSs, and some applications, support scripted batch files to automate tasks. Windows Powershell, Applescript, and Unix shell scripts are such scripting languages. They make complex or often-repeated tasks simple, especially for unskilled users.

THINK ABOUT COMPUTING 6.14
Python is a widely used scripting language, and well worth learning. You can download a Python interpreter and investigate it.

```
@echo off
echo Moving torrent files from c:\down and p:\torrents to Revo (Y:)
if exist c:\down\*.torrent     copy c:\down\*.torrent     y:\down
if exist p:\torrents\*.torrent copy p:\torrents\*.torrent y:\down
timeout 4
echo.
echo Done
```

FIGURE 6.19 A Microsoft DOS batch file

Modular programming

Modular programming involves breaking programs into small sections of code. Large programs are typically created as a collection of small, self-contained **code modules** (also known as subroutines or subprograms) for these reasons.

1. It is easier to find a bug in a small module than in a massive chunk of code.
2. Software development is faster when several programmers can work simultaneously on different modules.
3. A useful module can be re-used in other programs. This saves time and effort.
4. A program often needs to carry out the same action in different places. Rather than repeating the code, it appears once and is called upon multiple times.

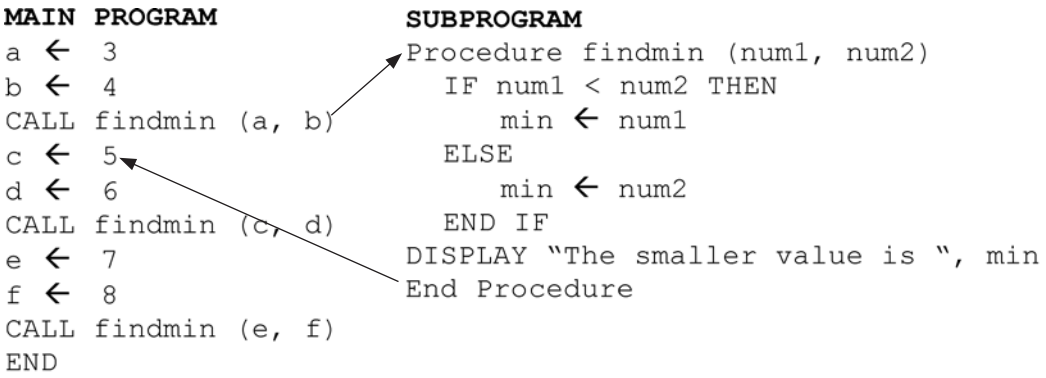


FIGURE 6.20 A main program calls a subprogram

There are a few notable features in Figure 6.20.

- The same code is needed in three places in the main program (left). Each time, the main program **calls** subprogram `FindMin`.
- For each call, the main program **passes** the subprogram two values it needs to calculate with. Values passed to a subprogram are called **parameters**.
- The values sent to the subprogram are copied to variables (`num1`, `num2`) that are **local** to the subprogram; that is, the main program cannot see or change them. In this way, a subprogram can use variables without worrying about variable names in the main program or other subprograms. **Global** variables are visible to – and changeable by – the main program and other subprograms. For safety, avoid using global variables unless absolutely necessary.
- When the subprogram finishes, it passes control back to the main program. Execution continues with the statement following the one that called the subprogram.

Functions

Functions are procedures that calculate and return a value. Function calls usually have parentheses after them, to contain parameters; for example:

```
answer ↗● SQRT(num)
```

Here, the `SQRT` function calculates the square root of parameter `num` and returns the answer to the main program where it is assigned to `answer`. Commonly used functions like `SQRT` come with the compiler for programmers to use. Typical function libraries include:

- mathematical: absolute, ceiling, cosine
- string: get left/right/middle characters, convert to lowercase, find substrings
- conversion: convert to Boolean/integer/string, convert string to number
- miscellaneous: time and date, logical (e.g. `isDigit`, `isUppercase`), random number.

Functions are often nested; for example:

```
PrintName ↗● Upper(Left(Firstname,1)) & "." &
Upper(Left(FamilyName,1)) & Lower(Right(FamilyName,Length
(FamilyName)-1))
```

Data validation

Validation checks that input data are reasonable. Validation does not and cannot check that inputs are accurate.

A **range check** checks that data are within acceptable limits or come from a range of acceptable values. For example, students enrolling in kindergarten must be between the ages of

THINK ABOUT COMPUTING 6.15
Visit a few websites that have data-entry forms. What **validation rules** are used on the data, and why? Are any of them unreasonable (e.g. insisting on a 5-digit zip code)?

3 and 6 years (acceptable limits). As another example, the product size must be small, medium or large (acceptable values).

A **type check** is a useful way of confirming that the values entered into fields are of the expected type. It will confirm if values are entered in the wrong fields, such as if numbers are entered into fields that expect only text values.

An **existence check** checks whether a value has been entered at all.

Internal documentation

Internal documentation explains the functioning and purpose of source code to programmers to make code more meaningful.

Useful comments add information that is not already obvious in the code:

```
// IntTemp - temperature is in Celsius
IntTemp ↗● 0
```

It should not be trivial or contain obvious information, like this:

```
// Set temperature to zero
IntTemp ↗● 0
```

What to include in internal documentation

- The purpose of a module
- The author's name
- Date of last modification
- Version number, to keep track of the latest version of the code
- Information, if any, about further work that is needed
- Problems that still need to be fixed
- Assumptions; for example, the customer file already exists
- Constraints; for example, it must work on a screen of only 300 × 500 pixels
- External code libraries or resources required by the module.

The contents of internal documentation may be dictated by a programming team's or organisation's style guide. It promotes consistency in a team's work, making it easier for programmers to collaborate and work with the code of others.

Adding internal documentation takes extra time and effort, but it is easier than studying obscure code.

There are no rules regarding how comments should be marked in pseudocode. You could use any of the following:

```
/* comment */
// comment
' comment
# comment
```

Note: Programmers are encouraged to write internal documentation, but for the purposes of this Area of Study, internal documentation is *not* required.

THINK ABOUT COMPUTING 6.16
Search online for 'google style guide'. Why do you think those rules were made?

Loops

Much of the power of software comes from the ability to automate repetitive actions. Using a loop – doing something 100 000 times – is just as easy as doing it twice.

If the number of required repetitions is not known in advance, use an **uncounted loop**; for example, when the number of accounts to process in a database might change every few minutes. It keeps testing whether it should continue looping.

If the number of required loops is known, use a **counted loop**; for example, the number of accounts has already been counted.

Counted loops

The classic counted loop is FOR/NEXT, which uses a variable as a counter (the **index**) to keep track of its progress as it loops from its starting point to its ending point.

This C code uses x as its index while looping from values 0 to 9 as it repeats the code within the curly { } braces to produce the output:

```
0   1   2   3   4   5   6   7   8   9
```

```
for ( x = 0; x < 10; x++ ) {
    printf( "%d\n", x );
    }
```

If you break down the typically concise C syntax, you can see the following.

- `x = 0` **initialises** the counter to value zero.
- `x < 10` looping continues while the value of x is less than 10.
- x++ increments x after each **iteration** (loop).

Inside the {loop}:

- printf is a function to display a formatted value.
- %d\n formats the output as a decimal followed by a new line.
- The semicolon tells the C compiler that the statement is finished.

This example in Basic is more typical of a real program because it uses variables rather than constants for the loop's limits.

```
FOR x = StartingValue TO EndingValue
    PRINT x
NEXT x
```

Uncounted loops

Uncounted loops keep cycling while a logical test is true; for example, *while* the temperature is less than 54 degrees Celsius, or *until* we reach the last **record** in the file.

Top-driven (test at top) loops carry out their continuation test *before* the loop begins. In contrast, **bottom-driven** (test at bottom) loops carry out their instructions at least once and then test to see if they should loop again.

TABLE 6.6 Uncounted loops

Top-driven loop (Test at top)	Bottom-driven loop (Test at bottom)

READ strName
Found ↗● FALSE
Pointer ↗● 1
WHILE NOT Found
  IF lstNames[Pointer] = strName THEN
  Found ↗● TRUE
  END IF
END WHILE
	READ strName
	Found ↗● FALSE
	Pointer ↗● 1
	DO
	  IF lstNames[Pointer] = strName THEN
	  Found ↗● TRUE
	  END IF
	LOOP WHILE NOT Found

Table 6.6 shows a number of classical programming features.

The spaces at the start of some lines are called **code indentation**. Code indentation makes it easier to see where loops and tests begin and end. You are expected to use indentation in your pseudocode.

Note also the **initialisation** of the `Found` and `Pointer` variables. Some compilers automatically initialise variables, but not all. Give variables explicit starting values, just to be sure. Close files explicitly and free up reserved memory before ending programs.

The `IF/THEN/END/IF` control structure is a classic example of programming logic.

Don't mix up '<' and '>'. It's embarrassing, and causes serious logical errors!

Logic

Digital logic is used to control a program's behaviour under different conditions. For example:

```
IF B > 0 THEN
    B ↗● B + 1
END IF
```

The logical test (`IF B > 0`) must result in a true or false (Boolean) answer. If true, the code following `THEN` is executed (`B ↗● B + 1`). If false, execution skips to the line after `END IF`.

The other logical comparisons in pseudocode are shown in Table 6.7.

TABLE 6.7 Logical operators

Symbol	Meaning
=	is equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
<>	is not equal to

Some logical decisions are more complicated than a single test.

Other logical operators can be used to allow powerful, intelligent decision making. Here are some examples.

- **AND** adds another condition that must also be true for the result of the test to be true.
  IF (ID length=5) **AND** (first character is alphabetical) the ID is valid.
  Both conditions need to be true, otherwise the whole **IF** statement becomes false.
- **OR** adds a condition that, if true, would make the whole test true.
  If (destination is far away) **OR** (time is short) then travel by jet.
  If *any* condition is true, the test result is true.
- **ELSE** describes what happens when the result of the **IF** test is false.
- **Parentheses** can be used to group tests into related logical bundles.

Take the example, 'An ID is valid if it is five characters long and starts with a letter, or it is seven characters long and starts with a digit'. The pseudocode for this might be as follows.

```
IF
            ((ID length=5) AND (first character is alphabetical))
            OR
            ((ID length=7) AND (first character is a digit))
THEN
            ID is valid
ELSE
            Display a warning.
END IF
```

- **CASE** is a handy structure used by many languages when there are many possible conditions, each with its own appropriate action.

Debugging

The best-laid plans of programmers can go astray in three main ways:

1 syntax errors
2 logical errors
3 runtime errors.

Syntax errors

Compilers expect precise instructions in a strict format with no ambiguity. If any source code cannot be understood by the compiler because the syntax does not match what the compiler expects, it will stop working until the code is fixed.

Such syntax errors are caused by incorrect punctuation, spelling and grammar. Incorrect punctuation may result from simple mistakes, such as using a square bracket instead of a parenthesis.

Incorrect spelling in a syntax error is not the same as incorrect spelling in regular communication. A compiler has a small dictionary of key words that it recognises. If a source code instruction is not in the dictionary, even if it is a real word, the compiler will return an error message. While a person may know that the word 'colour' has the same meaning as 'color', a compiler does not, so it returns an error until 'color' is used and 'colour' removed.

Commands in source code must follow a precise format to avoid incorrect grammar errors. For example, a language might expect the following syntax:

```
INPUT "prompt"; variablename
```

If the source code said the following instead:

```
INPUT variablename; "prompt"
```

the compiler would not understand the source code any more than you would understand a person who greeted you by saying, 'Like I hat your, hello!'

Thankfully, syntax errors are easily found by compilers, and easily fixed by programmers. In fact, many modern source code editors pop up helpful syntax tips when they detect the use of a key word, as shown in Figure 6.21.

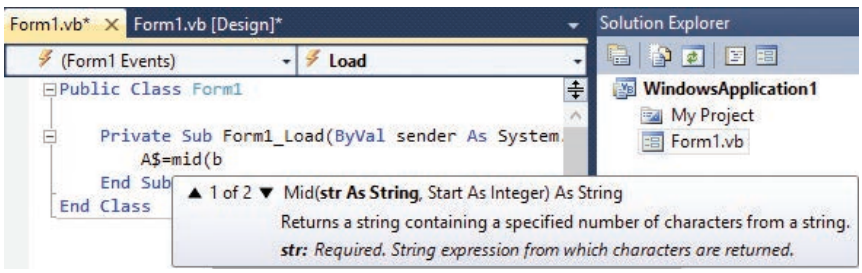


FIGURE 6.21 Visual Basic editor pops up help when you type a command

Logical errors

Logical errors occur when a programmer uses a wrong strategy. To the compiler, nothing is wrong with the syntax. The problem is that the instructions are just plain *wrong*. For example, to add 10 per cent tax to a price, the following statement would give an answer, but it would be wrong.

```
TotalCost = Price + 10%
```

A `Price` of 30 would yield a `TotalCost` of 30.10 instead of 33.00. What went wrong? The **algorithm**.

To calculate a price plus tax, the correct algorithm is:

1 Calculate 10 per cent of the `Price`.
2 Add that amount to the `Price` and store it as `TotalCost`.

The faulty algorithm simply added 10 per cent (0.10) to the price. It should have been:

```
TotalCost = Price + (10% of Price)
```

Logical errors are the hardest to fix because the compiler cannot detect faulty logic any more than a car knows when you are driving in the wrong direction. The only way to find logical errors is to create test data and manually calculate the correct answers for that data. Compare the algorithm's answers with the expected answers. If they do not match, fix your algorithm.

It is important to be aware of the potential for introducing incorrect spelling syntax errors because compilers use American English and Australian students write in Australian English.

Testing is discussed on page 248.

THINK ABOUT COMPUTING 6.17
Write an algorithm to convert a fraction like $\frac{3}{4}$ to a percentage.

Runtime errors

Runtime errors are caused by factors during the execution of a program, such as:

- the computer running out of memory
- hard disk errors
- operating system failure
- a problem with network connectivity
- incompatibility with another program running on the computer, such as antivirus software
- incorrect or outdated device drivers.

An OS may detect and handle unexpected system states to protect a program from crashing, but you should try to anticipate possible runtime problems and make allowances for them; for example, by making your code check for free space *before* saving data to disk.

The loop pseudocode (Table 6.6) introduces a common but unwelcome programming feature: a logical error. See if you can identify it before reading on.

The problem is that the program's loop uses the `Pointer` variable to count its progress through the array `lstNames`, but `Pointer` never changes and `Found` will never become `True` so the looping will never end. Such an **endless loop** will force the user to shut down the program to regain control of the computer.

To fix the logic, `Pointer` needs to be **incremented** (increased by one) with `Pointer ↗● Pointer + 1`. But where should the statement go?

Let's put the increment statement here:

```
Pointer ↗● 1
WHILE NOT Found
    IF lstNames[Pointer] = strName THEN
            Found ↗● TRUE
            Pointer ↗● Pointer + 1
    END IF
END WHILE
```

Is the code now debugged? We can find out by doing a desk check.

Desk checking

Desk checking is a technique used to check the logic of an algorithm manually. Essentially, the programmer imitates a compiler and manually tests pseudocode logic by stepping through the lines of code to check that the values are as they should be at each point. Throughout the check, you must adhere to specified logic. In the table below, assume that the name being searched for is 'Ted' and the `lstNames` array contains the following test data.

Array index	Value
1	Bob
2	Carol
3	Ted
4	Alice

Use your brain as a compiler to step through the code using our test data to calculate actual values. Beside the lines of code, draw a table where you record the values of variables.

This may help you remember the errors:
1 Syntax error: Trying to board a bus through the exhaust pipe.
2 Logical error: Getting onto the wrong bus.
3 Runtime error: The bus breaking down.

Remember that a logical error is a fault in the program's logic and will not be detected by a compiler or debugger.

When they first see statements like `Pointer = Pointer + 1`, some young programmers exclaim, 'How can something equal itself plus one?' In most languages, '=' indicates assignment, and means 'Evaluate `Pointer + 1` and store the result back into `Pointer`.' Unfortunately, most languages also use '=' for logical comparisons like `IF X=1 THEN X=0` The first '=' means 'is equal to'. The second means 'is assigned the value'. This is why, to avoid confusion, VCE pseudocode indicates assignment with '↗●' `IF X=1 THEN X ↗● 0`

THINK ABOUT COMPUTING 6.18
List each loop structure supported by your chosen language, and categorise its type.

The first desk check in Table 6.8 uncovers a new problem: When the `IF` test fails, execution skips to the `END IF` line, which bypasses the new increment statement, and causes that endless loop again! In the second desk check, we can move the increment statement again.

TABLE 6.8 First desk check

Code		Loop 1
READ strName	Strname = 'Ted'	
Found ↗● FALSE	Found = False	
Pointer ↗● 1	Pointer = 1	
WHILE NOT Found	Found = False, so test is True, so enter loop.	
IF lstNames[Pointer] = strName THEN		Pointer = 1. lstNames[1] = 'Bob'. 'Bob' <> 'Ted' so skip to END IF
Found ↗● TRUE		Skip
Pointer ↗● Pointer + 1		Skip
END IF		What? Hang on! That's NOT RIGHT.
END WHILE		

TABLE 6.9 Second desk check

Code		Loop 1	Loop 2	Loop 3	Loop 4
READ strName	Strname='Ted'				
Found ↗● FALSE	Found = False				
Pointer ↗● 1	Pointer = 1				
WHILE NOT Found	Found = False, so enter loop.		Found=False so loop again	Found still =False, so loop again	Found = TRUE! Skip to line after END WHILE
IF lstNames[Pointer] = strName **THEN**		Pointer=1. lstNames[1] = 'Bob'. 'Bob' <>' Ted' so skip to END IF	Pointer=2. lstNames[2] = 'Carol'. 'Carol' <> 'Ted' so skip to END IF	Pointer=3. lstNames[3] = 'Ted'. 'Ted' = 'Ted' so drop to next line	skip
Found ↗● TRUE		Skip (Found=False)	Skip (Found=False)	Found ↗● True!	skip
END IF					skip
Pointer ↗● Pointer + 1		Pointer=2.	Pointer=3	Pointer = 4	skip
END WHILE		Back to the top	Do it again	Back to the loop test again	skip

THINK ABOUT COMPUTING 6.19
Use the leap year pseudocode from earlier to desk check the years 1400, 1700 and 2100.

We expected the pseudocode to find 'Ted' in slot 3 of `lstNames`. The second desk check verified it.

Good test data caters for all possible circumstances, so you should also test that the code works when the name being sought is *not* in the array; for example, `strName` is 'Humphrey'.

Process testing

You must always test the software you have developed, whether the program is a game, a shopping cart for a website, or an aeroplane's autopilot navigation system. If the game you have developed fails, it may simply annoy the users. If the shopping cart fails, it could prevent purchases or overcharge customers. However, if the software in an aeroplane's autopilot system fails, people could die.

In addition to all of this, buggy, poorly tested code can ruin a freelance programmer's reputation and make it difficult to find future work.

Several forms of testing can be conducted.

Alpha testing, which is sometimes called informal testing, is when programmers test their own code during software development.

Beta testing is usually the first time that software is tested by future 'end users' and specially chosen reviewers, using live data that is more random in nature. It usually happens once the program has been completed and the goal is to weed out useability problems – you are actually trying to crash the software.

Validation testing verifies that the code properly validates input data. For example:

```
IF (Sex <> "M") AND (Sex <> "F") THEN
    DISPLAY "Invalid Gender!"
END IF
```

In **component testing**, individual modules within the software are tested in isolation. In contrast, **integration testing** is used to test whether modules work together; that is, that they can exchange parameters properly. **System testing** checks that the program works as a whole.

Finally, in the **formal testing** phase, the client who is paying for the software is shown how the finished program meets all of the functional and non-functional requirements specified during analysis.

Test data

Never trust a program's output. It may look perfectly authoritative but still be 100% wrong.

To prove the accuracy of a program's output, you need to feed the program sample data to work on, and compare the program's answer with one you know is guaranteed to be correct. Choosing this test data is not as easy as it sounds.

Good test data should include the following.

- Valid data, which is data that is perfectly acceptable, reasonable and fit to be processed
- Valid but unusual data, which is data that should *not* be rejected even though it seems odd; for example, a gifted 12-year-old child may enrol in university so that age number should not necessarily be processed as an error
- Invalid data, to test the code's validation routines; for example, if people must be 18 years of age to be granted a credit card, the test data should include people under 18 so they can be rejected
- Boundary condition data, which are on the borderline of some critical value where the behaviour of the code should change; such 'tipping point' errors are a frequent cause of logical errors in programming

Testing tables

Part of software design is to write a checklist of all the input, processing and output the software should be able to do, based on the design specifications. This list is used throughout the development of the application to check and test that the application meets those specifications. This list is called a testing table.

As an example, an online club allows members aged between 6 and 16. The validation pseudocode says:

```
IF age > 6 AND age < 16 THEN accept member
```

Is there a problem with this logic? You can use a testing table to use the test data to calculate a result manually and compare it with the output of the pseudocode. It is a good way to show evidence of testing in your Outcome.

In this case, good test data would be 5, 6, 7, 15, 16 and 17, because they cover every possible type of input: below the lower limit, on the lower limit, within the limits, on the upper limit, and above it. The test data set is as small as it can be.

TABLE 6.10 Testing table 1

Testing table			
Data	**Expected result**	**Actual result**	**Fix**
5	Don't accept	Don't accept	
6	Accept	Don't accept	
7	Accept	Accept	
15	Accept	Accept	
16	Accept	Don't accept	
17	Don't accept	Don't accept	

The pseudocode behaves accurately most of the time, but on occasion it fails spectacularly. Why? It only accepts members who are *over* 6 and *less than* 16 and rejects applicants who are *exactly* 6 or 16.

Once these logical failures have been highlighted, the cause of the errors need to be found. The rule effectively says 'between 6 and 16' not 'older than 6 and younger than 16'. The logical operators > and < are wrong. Now we need to devise a fix and fill in the last column of the testing table.

TABLE 6.11 Complete testing table

Testing table			
Data	**Expected result**	**Actual result**	**Fix**
5	Don't accept	Don't accept	🕑
6	Accept	Don't accept	Change Age>6 to Age >=6
7	Accept	Accept	🕑
15	Accept	Accept	🕑
16	Accept	Don't accept	Change Age<16 to Age <=16
17	Don't accept	Don't accept	🕑

Using the completed testing table in Table 6.11 as a guide, the pseudocode can be corrected as follows.

```
IF age >= 6 AND age <= 16 THEN accept member
```

Handy tip: Often when you are asked to identify a bug in code, the problem is a result of boundary condition errors.

Arrays

A variable can only hold a single value, which is a major limitation when processing a large amount of data. Adding up annual rainfall using variables looks like this:

```
GET Rain01
GET Rain02
GET Rain03
GET Rain04
GET Rain05
GET Rain06
GET Rain07
GET Rain08
GET Rain09
GET Rain10
GET Rain11
GET Rain12
TotalRain = Rain01 + Rain02 + Rain03 + Rain03 + Rain04 + Rain05 +
Rain06 + Rain07 + Rain08 + Rain09 + Rain10 + Rain11 + Rain12
```

Elegant code is effective, short and clever (also known as 'a neat hack'). Especially good algorithms are sometimes described as 'beautiful', and are more art than science.

Imagine the pain involved if there were 1000 rainfall figures. The solution is to use an array: a storage structure with multiple, numbered storage slots. Arrays used with loops are a powerful programming tool.

Using loops and arrays, the rainfall calculation could become far more elegant:

```
nMonths ↗● 12
DECLARE ARRAY Rain[nMonths]
FOR monthnum = 1 to nMonths
          GET Rain[monthnum]
          TotalRain ↗● TotalRain + Rain[monthnum]
NEXT monthnum
```

Using `nMonths` avoids repeating '12' throughout the code. The code is easier to maintain when key values can be changed in a single location.

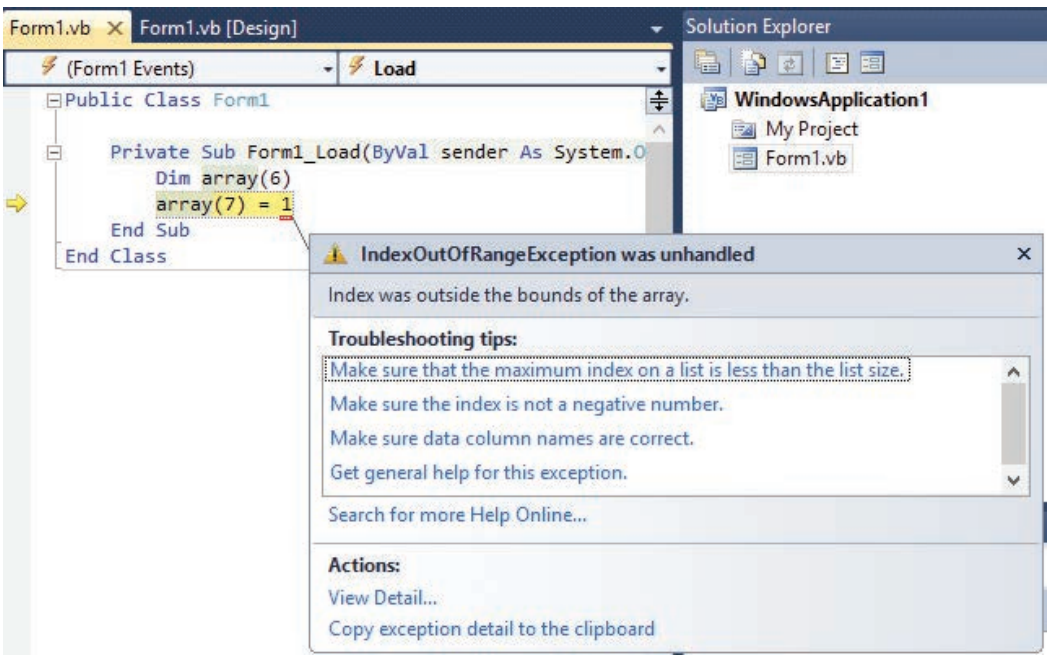


FIGURE 6.22 An attempt is made to refer to slot 7 of a 6-slot array

To cover 1000 months instead of just 12, simply change '12' to '1000' in the first line. It is that easy.

To create (or **declare**) an array, a programmer must usually specify the data type it will hold and the number of slots (or **elements**) it has. Referring to indexes outside of the defined limits will cause a runtime error.

The `Rain` array is a **one-dimensional (1D) array** because it is a single column of data, similar to a list. To store data in a table with rows and columns like a spreadsheet, use a two-dimensional (2D) array.

If the rainfall data needed to be stored for 100 years, you would create the array `Rain[100,12]` with 100 rows, each with 12 columns – 1200 slots all together.

As a table, the rainfall data may look similar to Table 6.12.

Some languages use [square brackets] to enclose array index numbers. Others use (parentheses). In pseudocode you can use either, but be consistent.

Arrays are much like tables. With tables, it often does not matter if you create it like this

	A	B
1		
2		
3		

or like this.

	1	2	3
A			
B			

Arrays are similar. We could have created the array as `Rain[12,100]`.

TABLE 6.12 A 2D array seen as a table

Array table								
	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6	Etc.	Month 12
	Jan	Feb	Mar	Apr	May	Jun		Dec
Year 1	67	34	67	45	34	45	...	67
Year 2	56	45	75	41	75	32	...	61
Year 3	41	63	82	46	56	31	...	86
Year 4	59	51	74	31	56	78	...	78
Etc.	...	...	...	...	...	...	...	...
Year 100	45	34	67	50	56	45	...	89

Having two dimensions requires two loops: one to loop through the 100 rows and one to loop through each row's 12 columns:

```
nMonths ↗● 12
nYears ↗● 100
DECLARE ARRAY Rain[nYears , nMonths]
// remember that years are the FIRST dimension, Months are the SECOND.
FOR yearnum = 1 to nYears
    FOR monthnum = 1 to nMonths
        GET Rain[yearnum , monthnum]
        TotalRain ↗● TotalRain + Rain[yearnum , monthnum]
    NEXT monthnum
NEXT yearnum
```

For each repeat of the outer year loop, the nested month loop carries out its full 12 cycles – like the hours and minutes on a digital clock. When the inner loop has run its 12 cycles, the outer loop increments its counter and the inner loop runs its full course again.

So the data is processed in the order shown in Table 6.13.

TABLE 6.13 How nested loops work

Year	Month	Comment
1	1	Inner and outer loops begin with their starting values
1	2	Outer loop's counter is constant until inner loop finishes its full count
1	3	Monthnum = 3
1	4	

Cont.

Year	Month	Comment
1	Etc.	
1	12	Inner loop finishes. Outer loop ticks over.
2	1	Inner loop starts its next full cycle.
2	2	Yearnum = 2. Monthnum = 2.
Etc.	Etc.	
100	10	
100	11	
100	12	Inner loop finishes. Outer loop finishes.

Important – an inner loop must be *completely* enclosed within the outer loop!

What if each of these rainfall figures for 12 months over 100 years were recorded in five different locations? Use a 3D array, `Rain[5,100,12]`, and three nested loops.

THINK ABOUT COMPUTING 6.20
Can you suggest a case when a 4-dimensional array would be needed?

Stacks

A **stack** is a simple, basic method of temporarily storing data, with the PUSH command, and releasing it as needed, with the POP command. Like a stack of pancakes or parked shopping trolleys, the most recently added item is always the first to be removed, so stacks are referred to as **first-in last-out (FILO)** or **last-in first-out (LIFO)** structures.

Stacks and queues are required knowledge for Software Development VCE Units 3 & 4.

Queues

Queues are often used when some code or a device such as a printer cannot keep up with incoming data or commands. Instead of discarding the code or device, they are stored in a queue in the order in which they arrived. When the processor or device becomes available, queued jobs are processed in turn. A queue is a **first-in first-out (FIFO)** stack.

Files and records

Primary storage, or RAM, holds data during a program's execution. Between runtimes, data and instructions are stored in files on non-volatile secondary storage, such as hard disk drives.

The main file types you should become familiar with are **sequential** and **random**.

Sequential files

Sequential (or serial) files are plain text documents that contain human-readable text data. Sequential files are easy to create but can be slow to search through because data items can be of any length. This means that the location of a particular data record may not be easily found.

A record is a complete set of data **fields** relating to a single item, person, transaction or event. An employee's record may include fields containing her given name, family name, date of birth, ID, department and so on. The concept of records and fields is also used in database theory, so it is worth remembering.

Sequential files are often used to store logs – ongoing histories of events. New events are easily appended (added to the end of) sequential files. OSs continuously maintain many logs.

A common type of sequential file is called CSV, which stands for comma-separated values. In a CSV file, each line is a record, composed of comma-separated fields:

```
Susan , Wasib , 1990/12/31 , WAS0001, Accounting
Spiro, Papadopolous, 1966/02/13, PAP0023, Management
Alphonse, Capone, 1924/11/01, CAP0003, Security
```

Random files

Random files are made up of records of identical length, with fields that must be rigidly defined in advance. Table 6.14 shows an example of a random file for an employee record.

Every record is the same length, so the location of any record in the file can be calculated precisely with:

```
StartingPoint ↗● (RecordLength * (RecordNumber - 1)) + 1
```

For example, the length of an employee record is 52 bytes. The starting point of record 3 in that random file would be (52 × 2) + 1 = 105. Using this calculation, the program can instantly seek position 105 in the file and load record 3.

Thus, any record can be accessed instantly without having to step through every record from the start of the file, as with sequential files.

Therefore, the benefit of random files is fast access. However, there are also some drawbacks to consider.

Data that is longer than the reserved file size is simply truncated (chopped off) and lost. 'Marybelle-ChristieAnne' would forever be known in the employee file records as 'Marybelle-Chris'.

A field value that is shorter than the allotted length still uses the allotted space. Thus, employee 'Su' would have 13 wasted bytes in her `FirstName` field on disk. With millions of records like these, a great deal of disk space can be wasted.

Try to imagine random files as books where every individual paragraph is allotted one exact full page of space. If you want to find paragraph 99 it is very easy: go to page 99. If paragraph 99 should be longer than a single page, the extra information is truncated and lost forever. If paragraph 99 is very short, most of the page is wasted.

Locating paragraph 99 in a normal book is much more like locating a sequential file. You must start at page 1 and count through 99 paragraphs. It takes more time but no information has been lost or space wasted.

Choosing between random and sequential files is not always easy, and it depends how much data there is, and how important it is to be able to access data quickly.

TABLE 6.14 A random file

Employee record	
Field	**Length (in bytes)**
`FirstName`	15
`FamilyName`	25
`DateOfBirth`	3
`ID`	7
`SickDays`	2

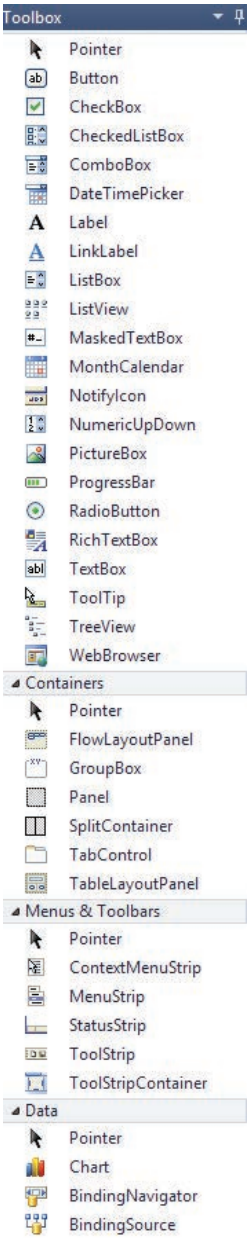


FIGURE 6.23 Visual Basic GUI Objects

GUI controls and structures

Most high-level languages have graphical user interfaces to make programs more intuitive for users.

These languages provide pre-packaged classes of objects for programmers to use, such as list boxes, menus, radio buttons and text boxes. With the GUI responsible for managing these objects, the programmer's work is made much quicker and easier.

Programmers can use GUI objects to avoid mastering arrays, variables, stacks and other programming basics, but this is short-sighted, because their programs will be tend to be fat and slow.

GUI structures are designed for ease of use, not raw power or optimal speed, and they consume far more memory and processor time than arrays, stacks and variables. The programmer also has little fine control over them, and essentially they are like cheap automatic cameras that are capable of producing acceptable, but rarely exceptional, photographs.

As a programmer, you must learn basic concepts, practise your skills, and master your tools to become successful.

When RAM was rare and very expensive, software had to be very basic, so it was difficult to use and difficult to create. Complex commands had to be memorised and usually typed. Software did not have the resources (RAM and hard-disk space) for all of the user-friendly features taken for granted now, such as easy menus, dropdown lists, cut and paste, undo, icons, dialogue boxes and scrollbars.

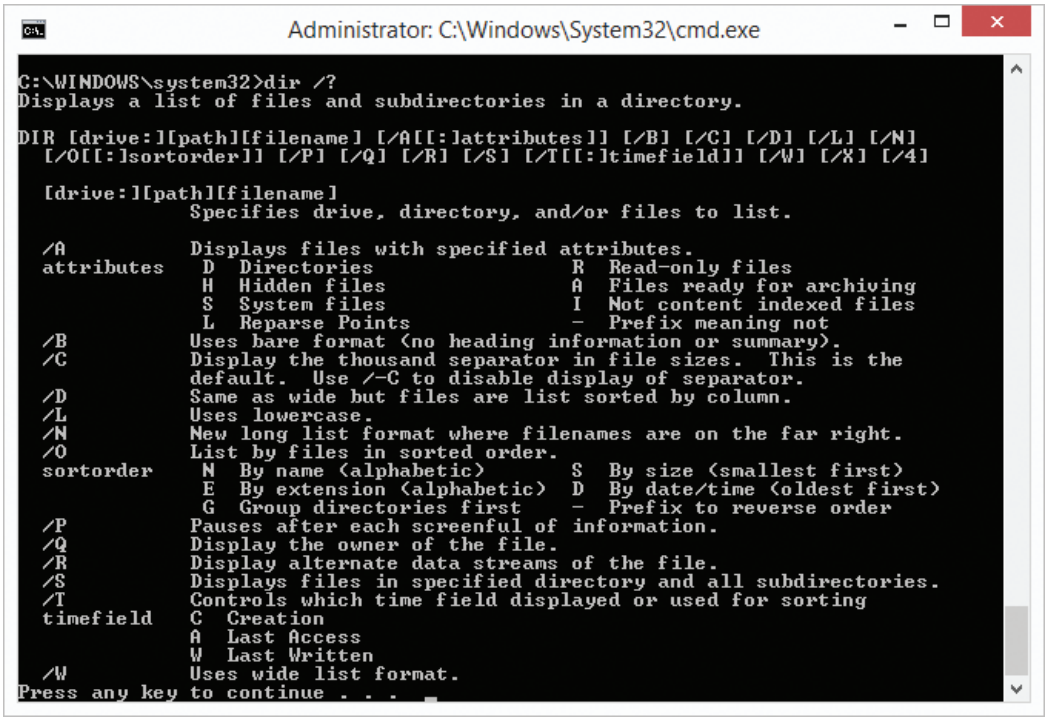


FIGURE 6.24 The unfriendly but powerful Windows command-line interface (CLI)

Eventually, researchers developed the concept of a graphical user interface where text commands could be replaced by intuitive actions such as clicking on icons and dragging objects around on screen. Credit for this 'WIMP' (Windows, Icons, Menus, Pointer) concept must go to Xerox PARC researchers from Palo Alto, California, who developed it in 1973. Apple later popularised WIMP ideas with the Macintosh in 1984. Microsoft and others soon followed.

GUI make software easier for end-users not interested in becoming power users. Power users are generally experts who have learned various efficient techniques, such as keyboard shortcuts and command-line text-based instructions.

GUI programming requires an object-oriented language that provides predefined classes (objects), such as windows, buttons and listboxes, with offer properties and methods, and which can respond to events.

The programmer writes code that can:

- read properties; for example, current mouse position
- change properties; for example, increase window height
- respond to events, such as a timer or a mouse drag
- cause an object to carry out a method, such as close, or refresh.

Object-oriented programming (OOP) lets programmers hand off a lot of work to the OS, such as detecting events like mouse clicks, sorting listbox items and scrolling windows. The programmer creates **event handlers**, which are procedures to handle events, such as, 'When the user clicks this button, take this action ...'

This type of programming is considered **event-driven** because the OS detects and reports events and the programmer provides responses to them. A drawback of GUI programming is that it creates programs with large appetites for RAM, disk storage and CPU power. Such programs may run poorly, or not at all, on mobile devices.

Searching

Finding a single item among billions, like a database of eBay auctions, is the sort of thing programmers constantly need to do, and need to do efficiently; that is, without wasting time, money and effort.

A **linear search** checks every individual item in turn to see if it matches the item for which you are searching. It can be very slow when there is a lot of data to search, but it is easy to program.

The faster, more clever search is the **binary search**, which is preferable when searching large data sets, especially when speed is important. The drawback is that the data set must be sorted, and sorting is itself a slow, difficult task.

There are many other types of searches, each with benefits under certain circumstances. Entire treatises, such as *The Art of Computer Programming* by legendary Donald Knuth, have been written to study the theory of searching.

CASE STUDY ETHICAL DILEMMA

To help you tackle this dilemma, search online for 'ACS code professional conduct' and look at the ethical requirements, particularly in Section 1, that the Australian Computer Society expects its members to follow.

Jean, a statistical database programmer, is trying to write a large statistical program needed by her company. Programmers in this company are encouraged to write about their work and to publish their algorithms in professional journals. After months of tedious programming, Jean has found herself stuck on several parts of the program. Her manager, not recognising the complexity of the problem, wants the job completed within the next few days. Not knowing how to solve the problems, Jean remembers that a co-worker had given her source listings from his current work and from an early version of a commercial software package developed at another company. On studying these programs, she sees two areas of code that could be directly incorporated into her own program. She uses segments of code from both her co-worker and the commercial software, but does not tell anyone or mention it in the documentation. She completes the project and turns it in a day ahead of time.

Source: a case study from Australian Computer Society: ACS Code of Professional Conduct Case Studies, April 2014.
More case studies: https://www.acs.org.au/__data/assets/pdf_file/0004/30964/ACS_Ethics_Case_Studies_v2.1.pdf
Code of professional conduct:
https://www.acs.org.au/__data/assets/pdf_file/0014/4901/Code-of-Professional-Conduct_v2.1.pdf

THINK ABOUT COMPUTING 6.21

Without reference to copyright or other laws, identify the following.

- What is Jean's ethical dilemma?
- What are her options?
- For each of these options, what are the likely consequences of her choosing it?
- How could she get out of this dilemma?
- What relevant clauses in the 'Australian Computer Society's Code of Professional Conduct' cover this situation?

Ethical dilemmas are discussed in Chapters 1 and 5.

CHAPTER SUMMARY

ESSENTIAL TERMS

accessibility ease of use by people with disabilities or special needs

algorithm strategy behind a calculation or procedure

arrays storage structures with many 'slots' (elements) that are addressed by number; arrays are managed with loops

central processing unit (CPU) the 'brain' of a digital system; the handler of most of a system's data manipulation

code modules subroutines or subprograms; a function is a special type of module because it sends back (returns) a value; programmers must keep track of storage structures in multiple modules, making data dictionaries even more valuable

communication hardware hardware that transmits data between computers and networked devices

compiler programs that convert source code into executable programs

debugger a program that helps to remove programming errors (bugs)

default action the operation that the software will carry out when the user does not give more detailed instructions

default value used if the user does not provide an alternative value; a word processor may default to using Arial typeface, 12pt, with single-spacing

editor a specialised word processor for creating human-readable programming instructions

field a single data item in a record; e.g. FamilyName

graphics processor unit (GPU) a very fast and expensive processor specifically designed for high-speed image processing in graphics cards

initialise to give a starting value to a variable

input devices instruments and peripherals, such as keyboards, that allow users to give data and commands to software and the OS

integrated development environment (IDE) a unified programming tool

interface within software, the place where people control the program, enter data and receive output

iteration looping or repeating

linker a program used to load information that the executable code will need, such as read a keyboard or calculate square roots

methods actions a GUI object can carry out; e.g. window.refresh.

mock-up a sketch showing how a screen or printout will look, which is used to aid in the design of an interface

modular programming breaking programs into small sections of code to simplify debugging and allow the re-use of modules in other programs

object any item that a program can inspect and/or change, in terms of appearance, behaviour or data

operating system (OS) software programs that manage a computer's hardware and run programs

output devices instruments and peripherals, such as printers and monitors, that display information from a computer in human-readable form

platform a combination of OS and CPU

ports physical connectors (sockets) for cables

primary storage random-access memory (RAM), which provides storage for data, information and software during program execution

processing hardware (CPU, GPU) hardware that runs the operating system, utilities and applications

properties characteristics such as width, colour, visibility

prototype a demonstration product that looks and feels like a finished program, but may be incomplete or not fully functional

pseudocode code that designs algorithms in a clear, human-readable, language-independent format

random-access memory (RAM) the primary and most common form of hardware storage; it can be accessed randomly, meaning that any byte of memory is accessible without touching the preceding bytes

random files records of identical length with fields that must be rigidly defined in advance

reduced instruction set computing (RISC) CPUs (like ARM), which have smaller instruction sets than complex instruction set computing (CISC) CPUs

queue a 'First In First Out' stack, storing incoming data or jobs to be processed in order

record a complete set of fields relating to an entity, such as a person

secondary storage (HDD, SSD) permanent storage

sequential file plain text with variable field lengths, such as CSV; large files are slow to search

source code human-written and human-readable version of a program

CHAPTER SUMMARY

stack a simple, temporary 'first in last out' (FILO) storage structure

storage structures places in memory holding data that is being used by a program; includes variables, arrays, textboxes and radio buttons

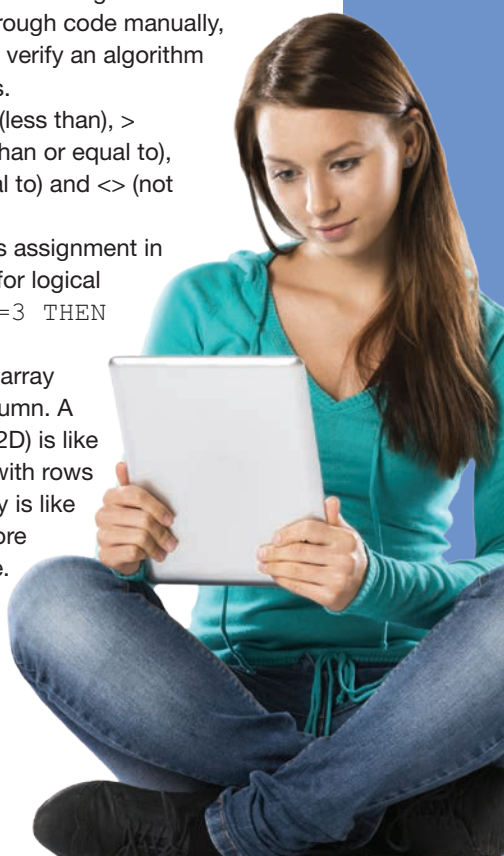
style guide in software development, a set of instructions for programmers about how to design software for the developer's platforms

thin client the belief that it is better to use 'dumb' workstations connected to a powerful central computer that does all the processing work, rather than use many powerful computers

validation rules rules that check the reasonableness (not accuracy) of data's range, type, and existence

IMPORTANT FACTS

1. The three types of software are system (OS), applications and utilities.
2. Interpreted languages convert source code to executable code only when a program runs.
3. Information systems are made up of people, data, processes and equipment (digital systems).
4. Digital systems comprise hardware, software, networks, protocols and application architecture patterns.
5. Transmission media such as twisted-pair cables, fibre-optic cables and WAPs each have their strengths and weaknesses.
6. The problem-solving methodology (PSM) is used to guide software development.
   - Analysis determines the software's requirements, scope and constraints.
   - Design devises a method of solving the problem, including its appearance and architecture. Tools include pseudocode, IPO charts, data dictionaries and mock-ups.
   - Development is when software and hardware are created, assembled and tested. Training and documentation are created.
7. Good naming practices include Hungarian notation (e.g. intTemp) and CamelCase.
8. A data dictionary lists all storage structure requirements, including data types and names, size, validation rules etc.
9. Data types include integer, floating point, string, character, date/time and Boolean (true/false).
10. IPO charts plan the data and processing needed to produce output.
11. Data structure diagrams describe the structure and relationships within complex data structures.
12. Object descriptions completely describe the properties of objects; e.g. textbox.width.
13. Internal documentation in source code explains the workings of the code to programmers.
14. Mock-ups show what screens and printouts should look like, including sizes and positions of items, colours, alignment and fonts.
15. Onscreen images are saved with RGB colour information. Printed images use CMYK.
16. Modules are called subprograms, procedures or subroutines. Functions are modules that return a value.
17. Global variables are visible to and changeable by the main program and all subprograms.
18. Local variables only exist within a single module.
19. Counted loops (`FOR/NEXT`) repeat a known number of times.
20. Uncounted loops (`DO`, `WHILE`) loop while a condition is true.
21. Uncounted loops can be top-driven (test at top) or bottom-driven (test at bottom), depending on where they test for continuation.
22. Syntax errors occur when a compiler cannot understand instructions in source code.
23. Logical errors occur when a faulty calculation strategy produces incorrect output.
24. Runtime errors occur when problems arise during execution, such as running out of RAM.
25. Desk checking steps through code manually, acting like a compiler to verify an algorithm and detect logical errors.
26. Logical operators are < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) and <> (not equal to).
27. The operator ↗● means assignment in pseudocode. = is used for logical comparison (e.g. `IF A=3 THEN B ↗● 4`)
28. A one-dimensional (1D) array is like a list with one column. A two-dimensional array (2D) is like a table or spreadsheet with rows and columns. A 3D array is like a stack of 2D arrays. More dimensions are possible.



CHAPTER SUMMARY

29 Data can be saved to secondary storage as sequential or random files.
30 Random files have fixed-length records that are quickly accessed.
31 Testing ensures that programs work properly and generate accurate information.
32 Good test data checks software behaviour under all possible circumstances.
33 Prove your algorithm and show evidence of testing by using a testing table.
34 Events include mouse clicks, timers and so on.
35 Object-oriented languages use classes of pre-defined objects (e.g. listboxes) that programmers can manipulate.

TEST YOUR KNOWLEDGE

Review quiz

HARDWARE

1 Describe two differences between primary and secondary storage.

SOFTWARE

2 Explain the difference between system and application software.

THE OS (OPERATING SYSTEM)

3 Summarise the role of an operating system.

4 Describe the roles of people, data, processes and digital systems in creating an information system.

PROGRAMMING AND SCRIPTING LANGUAGES

5 'Programming languages may differ in syntax, but they are all basically alike.' What does this mean?

SOFTWARE DEVELOPMENT TOOLS

6 How are source code and executable code related?

STORAGE STRUCTURES

7 Name and give examples of five data types.

8 Describe how lossy and lossless compression reduce file sizes.

THE SOFTWARE DEVELOPMENT PROCESS

9 Why is the PSM analysis stage important?

10 Show examples of three different software design tools.

11 What is an algorithm, and how does pseudocode relate to it?

12 Suggest two possible good names for a variable used to hold an integer value describing men's average shoe size.

CREATING EFFECTIVE USER INTERFACES

13 Describe affordance and tolerance and give examples of each.

14 List five tips for creating an effective user interface.

FUNDAMENTAL PROGRAMMING CONCEPTS

15 List examples of syntax, logical and runtime errors.

16 How do compiled and interpreted languages differ?

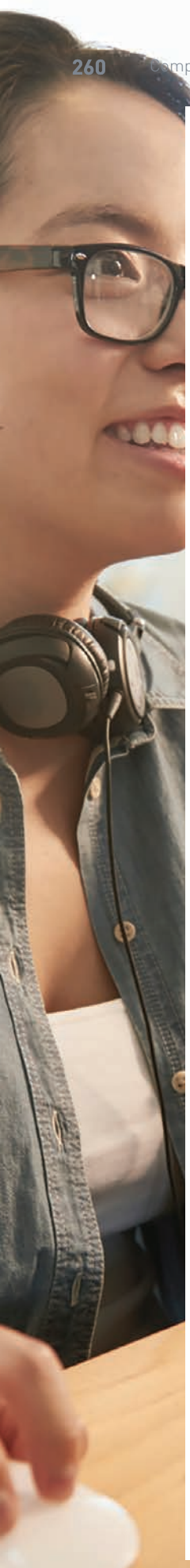
17 Justify the use of internal documentation.

18 'Arrays and loops were built for each other.' Explain.

19 When is desk checking used?

20 Why is it important to use test data that focuses on boundary conditions?

21 When would you recommend the use of a random file?



APPLY YOUR KNOWLEDGE

GUESSING GAME

You will design and develop a program that will play a game with a human. The human thinks of an integer between 1 and 100. Your program will use the most efficient strategy possible to guess the number. For each guess, the human must confirm if the guess was correct, higher or lower than the secret number.

Tasks

1. Write pseudocode to describe the processing strategies involved in developing the program.
2. Create an IPO chart to describe the data and information requirements.
3. Design the program's interface with a mock-up.
4. Create test data to fully exercise the validation and the algorithms.
5. Use your test data to desk check the pseudocode.
6. Develop the solution, using good object naming and creating useful internal documentation along the way.
7. Test the solution with your test data. Fix all bugs.
8. Get a classmate to play your game and give you written feedback on its appearance and ease of use.

PREPARING FOR

UNIT 2 OUTCOME 1

Design working modules in response to solution requirements, and use a programming or scripting language to develop the modules

STEPS TO FOLLOW

The Outcome will require approximately 10 classes to complete. For each programming module, you will be given solution requirements (the analysis part of the PSM), and you will select appropriate design tools to plan a solution, and then develop and test a working model of the solution. Use pen and paper for the design to avoid prematurely beginning development.

Your teacher may choose to give you the tasks one at a time after periods of relevant theory instruction, or as a group after all of the theory has been covered. To encourage serious attention to design, your teacher may choose to enforce design-only periods of time where no computers may be used.

Keep all of your designs, even failed attempts. A change of mind during design can actually be a very desirable event!

DOCUMENTS REQUIRED FOR ASSESSMENT

- Screen mock-up
- Data dictionary
- IPO chart
- Internal documentation (must be used within the source code)
- All solutions must show evidence of testing using appropriate test data.

ASSESSMENT

You will be assessed on the following measures.

- Whether your chosen design tools are appropriate and useful
- Your choice of data types and storage structures
- Object and file naming
- The accuracy of calculations
- The appropriateness of software functions you have used
- Functionality: Does the program do everything it is meant to do?
- Appearance of both the interface and the output
- Ease of use
- Internal documentation
- Thoroughness of testing