CONTENTS

Student guide

This teaching and learning material is designed to cover all the skills, knowledge and understanding that you need to pass the Software Development unit of Intermediate 2 Computing.

To achieve this unit, you must develop and demonstrate knowledge and understanding of:
• the principles of software development
• software development languages and environments
• high level language constructs
• standard algorithms

At the end of the unit, you will be tested on this by sitting a short 20- question multiple choice test.

However, it is not only about passing a test. You must also develop practical skills in software development using a suitable high level language. Almost any programming language can be used, but these notes assume that you are using Visual BASIC 5. If you are using a different programming language, your tutor will need to supply you with other materials for some parts of the unit.

Your tutor will complete a practical skills checklist for you as you work through the practical exercises in these notes. You should keep a folio of evidence; this should include documentation of all the stages of the software development process.

You will see the following icons throughout these notes:

Computer-based practical task – you will need access to a computer with Visual BASIC 5 (or similar) installed for this task**.**

Questions for you to answer – you can check your own answers against the sample answers given at the end of this pack.

Activity (non computer-based) – this will usually require some written work.

You should ask your tutor to check your work whenever you complete a computer-based practical task or a non computer-based activity.

SECTION 1

1.1 Computer languages

Just as there are many human languages, there are many computer programming languages that can be used to develop software. Some are named after people, such **Ada** and **Pascal**. Some are abbreviations, such as **PL/1** and **Prolog**. All have different strengths and weaknesses. **FORTRAN** was designed for carrying out mathematical and scientific calculations. **Prolog** is good for developing programs in artificial intelligence. **COBOL** is for developing commercial data processing programs.

Activity: Make a list of six or so programming languages (you can find these in textbooks or on websites). For each one, write down where it gets its name from, and what it is 'good' for.

Here are some examples to get you started:

Name	**Source of name**	**Used for**
Ada	after Countess Lovelace	US military systems
Logo	Greek for 'thought'	education
FORTRAN	FORmula TRANslation	early scientific language

All these languages are what we call **high level languages**. That is to distinguish them from **low level languages**! What do we mean?

1.2 High and low level languages

Inside every computer, there is a processor. This is a chip containing digital electronic circuits. These circuits work with tiny pulses of electricity and electronic components. The pulses of electricity can be represented by the digits 1 and 0. Every item of data, and every instruction for the processor is represented by a group of these binary digits.

Processors only 'understand' these binary digits. The only inputs you can make to a processor are groups of binary digits. The only output that a processor can make is a group of binary digits.

Instructions and commands made in these binary digital form for processors are known as **machine codes**.

Here are a few machine codes for a 6502 processor:

```
10101001 00000001
10000101 01110000
10100101 01110000
```

I'm sure you'll agree that they are not very easy to understand.

There are several problems with machine code:
• machine codes for different processors are different
• they are very hard for humans to understand and use
• they take up a lot of space to write down
• it is difficult to spot errors in the codes.

Unfortunately, processors don't understand anything else, so machine code has to be used. The earliest computers could only be programmed by entering these machine codes directly. It was a slow process, easy to get wrong, and it was very difficult to track down and fix any bugs in the programs. Machine codes are an example of **low level languages**, understood by the low level components of the computer system (the processor and other electronic circuits).

To get round these difficulties, computer scientists invented **high level languages**.

High level languages are similar to human languages. Instead of using binary codes, they use 'normal' words. For example, the computer language BASIC uses words like PRINT, IF, THEN, REPEAT, END, FOR, NEXT, INPUT and so on. That means that high level languages are easier to understand than machine code, and are more 'readable', that is, it is easier to spot and correct errors.

On the next page is a simple program written in a number of high and low level languages:

The first three are all examples of high level languages (BASIC, Logo and Pascal). All use words that are understandable to humans.

```
10 Number:= 1
20 Answer:= Number + 1
30 PRINT Answer
40 END
```

```
make 'number 1
make 'answer 'number + 1
say 'answer
```

```
PROGRAM adder;
VAR answer,number: real;
BEGIN
Number:=1;
Answer:=number+1;
WRITELN(answer);
END.
```

This is a low level language called 6502 assembler – not so easy to understand!

```
LDA #1
STA 1000
LDA 1000
ADC
STA 1001
JSR OSWRCH
RTS
```

And this final one is 6502 machine code, which is completely unintelligible to (most) humans.

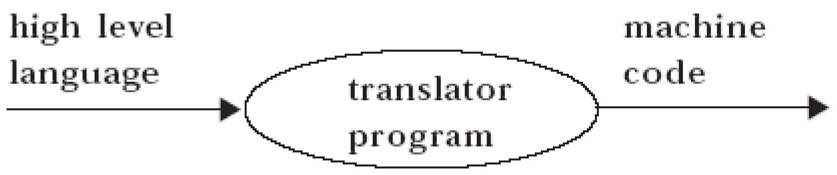
```
10101001 00000001
10000101 01110000
10100101 01110000
01101001 00000001
10000101 01110001
00100000 11101110
11111111 01100000
```

In fact, all five of these programs do more or less the same job! I think you will agree that high level languages are much more practical for writing programs than machine code!

Questions:

1. Which type of language (high or low level) is easier to understand?
2. Which type would be easier to correct if it had a mistake in it?
3. Name two low level languages.
4. Name two high level languages.
5. Explain the main differences between high and low level languages.
6. List two advantages of high level languages.

It looks as though high level languages have all the advantages compared to machine code. However, there is one major problem – processors don't understand high level languages at all! To get round this problem, computer scientists have developed translator programs which can translate high level languages (written by humans) into machine code (understood by processors).



1.3 Translators

There are two main types of translator program that you need to know something about. These are called **interpreters** and **compilers**.

To understand the difference, it is useful to think about an analogy from the 'non-computer' world.

Imagine that you are the world expert in some obscure subject, like 'the anatomy of the microscopic tube worms of the steppes of Kazakhstan'. You have been invited to present a lecture on this subject at a conference to be held in Japan. Most of the delegates at the conference do not speak or understand English, and you do not know any Japanese. How are you going to communicate?

There are two options.

- **Interpreters**
- **Compilers**

1.4 Interpreters

Option 1 is to go to the conference yourself, and deliver your speech in English one sentence at a time. After each sentence, a professional translator (who can understand English and also speaks fluent Japanese) will turn your sentence in Japanese. This will continue right through your lecture, with the interpreter translating each sentence as you go along.

Computer interpreter programs work in the same way. The interpreter takes each line of high level language code, translates it into machine code, and passes it to the processor to carry out that instruction. It works its way through the high level language program **one line at a time** in this way.

This works fine, but it has a couple of important disadvantages. Think about the analogy again. Your one-hour lecture will take two hours to deliver, as each sentence is spoken by you in English, then by the interpreter in Japanese. The other disadvantage is that if you are then asked to deliver your lecture again in another Japanese city, you will need to have it translated all over again as you are delivering it the second time.

The same problem is true of computer interpreters. The process of translating the high level language (HLL) program slows down the running of the program. Secondly, the HLL program needs to be translated **every** time it is used. This is a waste of computer resources and means that the user must always have an interpreter as well as the HLL program (often called source code).

1.5 Compilers

An alternative approach is to use a compiler.

Going back to the Japanese lecture example – instead of using a translator at the conference, you could write down the text of your lecture in English, and get a translator to translate it all into Japanese in advance. You could then send the translated lecture script to the conference, and have it read out by a Japanese speaker there.

The advantages are obvious – your lecture can be delivered in the one hour allowed in the conference programme, and it can be used as often as required without it needing to be translated over and over again.

A compiler program works in the same way. It takes your HLL program, and translates the **whole** program into machine code once. This machine code can then be saved and kept. Once translated, it can be used over and over again without needing to be translated every time. The compiled program therefore runs more quickly, and the user doesn't need to have a translator program on their own computer.

Software that you buy, such as a games program or an application, will have been **compiled** into machine code before being distributed and sold. What you get on the disk or CD is a machine code program that can run on your computer without needing to be translated.

Questions:

1. Name the two main types of translator programs.
2. Which one translates a whole program into machine code before it is executed?
3. Which one translates a program line by line as it is being executed?
4. Why do machine code programs run more quickly on a computer than high level language programs?

1.6 Text editors

During the development of a high level language program, after the analysis and design stages, the programmer (or team of programmers) have to **implement** the design by coding it in a suitable high level language.

Here is an example of a Visual BASIC program:

```
Private Sub cmdOK_Click()
‘ coding for the OK command button
‘ displays an appropriate message for each possible number entered
‘ written by A. Programmer on 29/12/03
Dim Number as Integer

Number = txtNumber.text

If Number = 1 Then MsgBox Number & “ wins you a colour TV”
If Number = 2 Then MsgBox Number & “ wins you a mobile phone”
If Number = 3 Then MsgBox Number & “ wins you a holiday in Spain”
If Number = 4 Then MsgBox Number & “ wins you 10p”
If Number = 5 Then MsgBox Number & “ wins you a day at the beach”
If Number < 1 Then MsgBox Number & “ is too small”
If Number > 5 Then MsgBox Number & “ is too large”
End Sub
```

You can see that a high level language has features that make it similar to a human language – the use of ordinary words, for example. This means that the implementation is often carried out using similar tools to those used for writing an essay or report. For example, cut and paste would be useful when typing the program shown above. To write an essay or report, you would normally use a word processing package. High level language programs can also be written using a word processing package. The **source code** can be saved as a text file, which can then be translated into machine code by a compiler.

However, some software development environments provide a **text editor** which incorporates many of the usual features of a word processor. The most useful of these is probably the ability to cut and paste sections of code.

Activity:

Consider the software development environment you are using for the programming section of this unit.

Does it have a text editor, or do you use a separate word processing package? What useful text editing features does it incorporate?

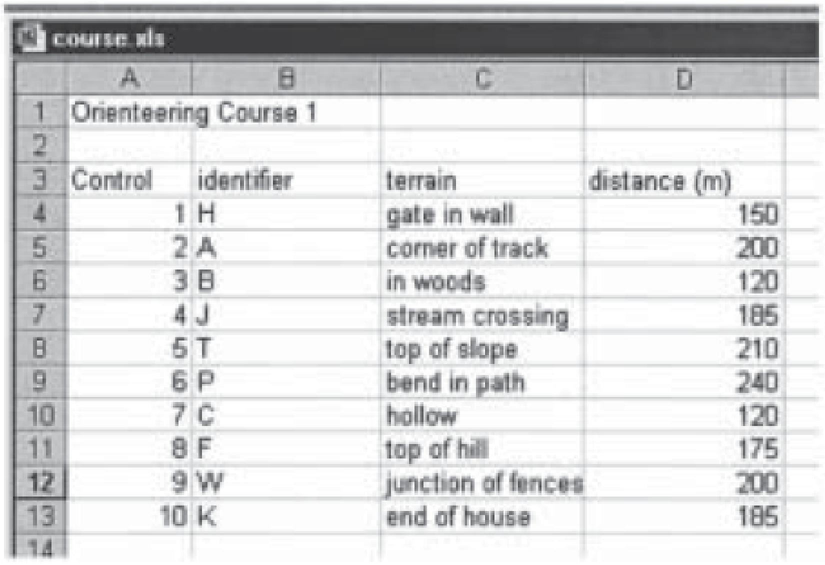
1.7 Scripting language and macros

Most of this unit is concerned with the process of developing programs written in a high level language to create stand-alone applications.

However, small programs called macros can be developed within some existing application packages.

Example 1: creating an Excel spreadsheet macro

Set up a small spreadsheet like this:



course.xls

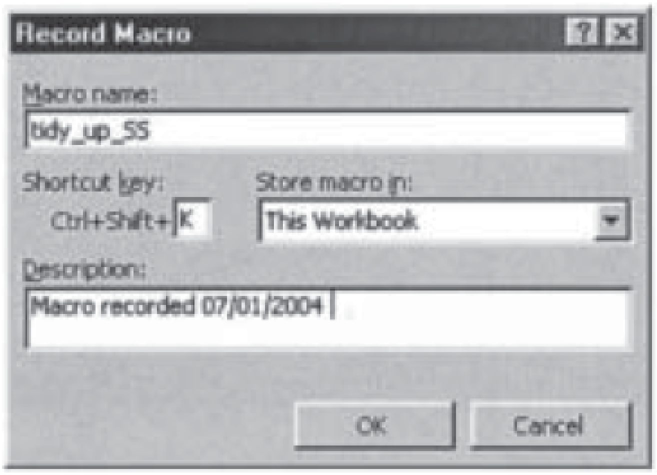
	A	B	C	D
1	Orienteering Course 1			
2				
3	Control	identifier	terrain	distance (m)
4	1	H	gate in wall	150
5	2	A	corner of track	200
6	3	B	in woods	120
7	4	J	stream crossing	185
8	5	T	top of slope	210
9	6	P	bend in path	240
10	7	C	hollow	120
11	8	F	top of hill	175
12	9	W	junction of fences	200
13	10	K	end of house	185
14				

Save it as **course.xls**

Save a ***second copy*** of the same spreadsheet as **course_copy.xls**

From the **Tools** menu, select **Macro**, then **Record New Macro**.

The following dialog box should appear:



Enter the name (**tidy_up_SS**)

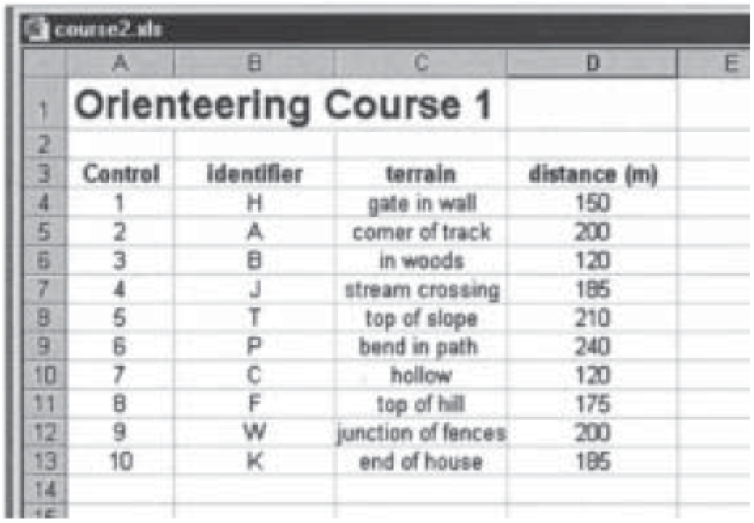
And the shortcut key (**Ctrl + Shift + K**)

Then click **OK**

Warning: follow these instructions very carefully – all your actions are being recorded!

- Select cell A1 (the title of the spreadsheet)
- Change its font to 18pt Bold
- Select A3 to D13 (all the data)
- Centre it all using the centre button on the menu bar
- Select row 3 (the column headings)
- Make them bold.

The spreadsheet should now look like this:



course2.xls

	A	B	C	D
1	Orienteering Course 1			
2				
3	Control	identifier	terrain	distance (m)
4	1	H	gate in wall	150
5	2	A	corner of track	200
6	3	B	in woods	120
7	4	J	stream crossing	185
8	5	T	top of slope	210
9	6	P	bend in path	240
10	7	C	hollow	120
11	8	F	top of hill	175
12	9	W	junction of fences	200
13	10	K	end of house	185

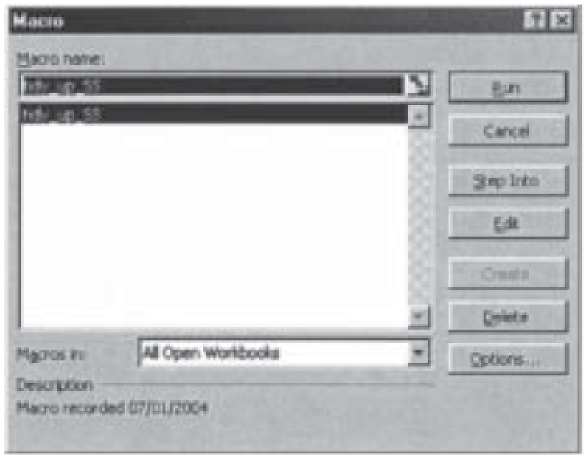
Click on **Stop Recording**

Save the improved spreadsheet as **course2.xls**

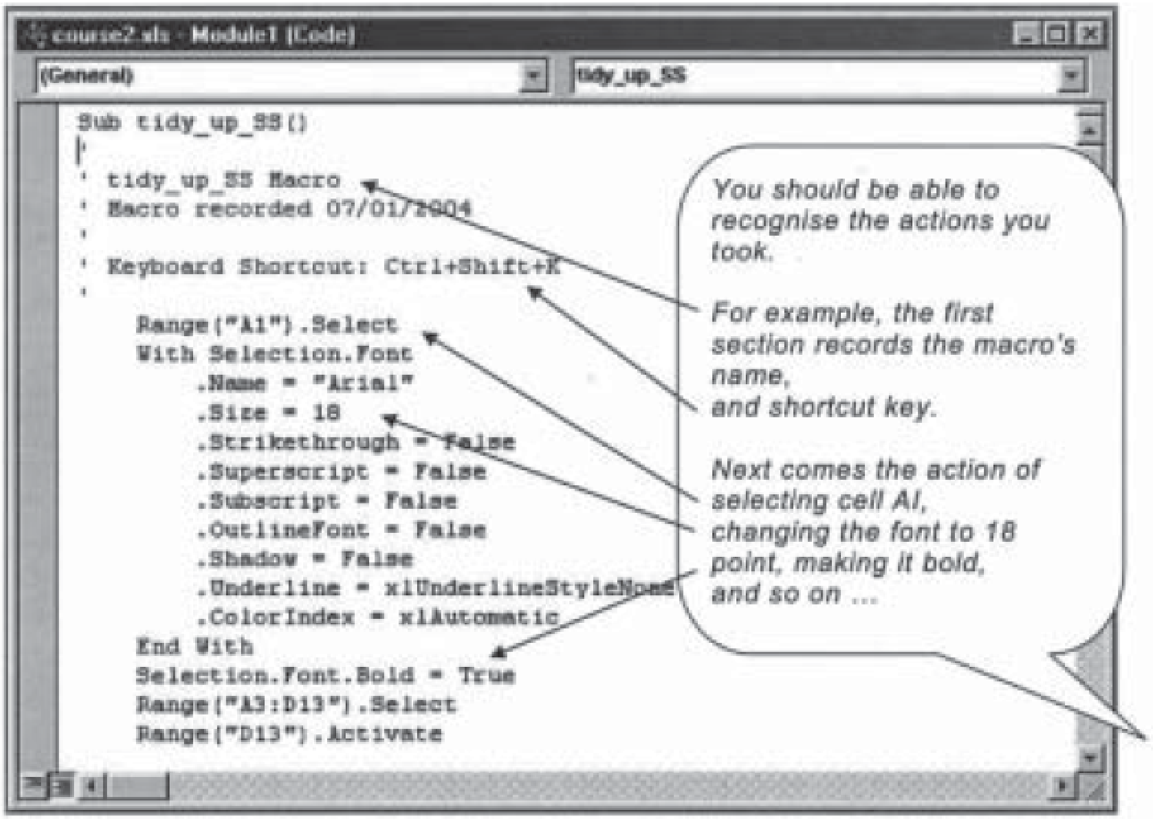
All the series of actions that you applied to **course.xls** to turn it into **course2.xls** have been recorded and stored as a **macro**.

To see the macro you have created:
• go to the Tools menu
• select Macro
• select Macros

A dialog box like this should appear, with your named macro listed under the name you gave it. Click on **Edit**.



Another window will open, which displays the code of the macro you have recorded, like this:



```
Sub tidy_up_SS()
'
' tidy_up_SS Macro
' Macro recorded 07/01/2004
'
' Keyboard Shortcut: Ctrl+Shift+K
'
    Range("A1").Select
    With Selection.Font
        .Name = "Arial"
        .Size = 18
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlUnderlineStyleNone
        .ColorIndex = xlAutomatic
    End With
    Selection.Font.Bold = True
    Range("A3:D13").Select
    Range("D13").Activate
```

You should be able to recognise the actions you took.

For example, the first section records the macro's name, and shortcut key.

Next comes the action of selecting cell AI, changing the font to 18 point, making it bold, and so on ...

The macro is actually coded in a **scripting language** called Visual BASIC for Applications, or VBA for short.

What use is a macro?

- keep **course 2.xls** open
- open **course_copy.xls**
- hold down **Ctrl + Shift + K**

The file **course_copy.xls** should be automatically formatted by the macro to be the

same as **course2.xls**.
If the user had several similar unformatted spreadsheets and wanted them all formatted in this way, he could save a great deal of time by using the macro.

A macro is a time-saving program written in a scripting language which can be activated by a series of key strokes for repeated use. A macro cannot exist alone – it only works with an application program (in this case, Excel). In this example, we have seen a macro being used with a spreadsheet. Macros can be used with many other application packages.

Example 2: creating a word processing macro

• open any word processing document
• as before, from the **Tools** menu, select **Macro**, then **Record New Macro**
• name the macro bold_red_text
• assign a shortcut key combination (perhaps **Ctrl + Alt + R**)
• click **OK** (Now the macro is being recorded)
• select bold and text colour red from the menu bar
• click to stop the macro recording.

Now you can use the macro.
• select any block of text
• activate the macro by using the shortcut key combination.

You can also activate the macro by selecting it from Tools, Macro, Macros.

This macro would be useful if you have several documents to work through, in each of which you have been asked to change the main heading to bold red text.

If you needed to change all the sub-headings to italic blue text, you could set up another similar macro to do that. Alternatively, you could edit the macro directly by changing the VBA code in the edit window. Try editing the above macro to make it produce blue italic text.

The examples above are very simple ones. Macros can be used to automate any task within an application program. For example, they can be used to activate long and complex data manipulations within a database application, or specialised formatting within any type of document.

Some applications, such as AppleWorks, allow you to record macros, but don't allow you to edit the code as you can in MS Office. If you have time, you could explore any other applications that you use, to see if they have a macro facility.

Questions:

1. What is a macro?
2. What type of language is used to write macros?
3. What are the advantages of using macros?
4. Describe two examples where a macro could be useful.

SECTION 2

2.1 Software

This unit is about **software**.

What **is** software?

You should already know that any computer system is made up of **hardware** and **software**.

The term **hardware** is fairly easy to understand, because you can see it. It is all the pieces of equipment that make up the system – the processor, monitor, keyboard, mouse, printer, scanner and so on.

Software is not so obvious. It is all the programs, instructions and data that allow the hardware to do something useful and interesting.

Think about all the different items of **software** that you have used in the last week or so.

Here is the list of programs that I have used recently:
- **Microsoft Word** (the word processing program that I use – I regularly use three versions of it: Word 2000, Word 98 for MacOS 8, Word v.X for MacOS X)
- **Microsoft Excel** (spreadsheet used to keep charity accounts for which I am the treasurer)
- **ClarisWorks 4** (integrated package – I mainly use its word processor and simple database sections)
- **Internet Explorer** (both PC and Mac versions – for browsing the web)
- **Safari** (web browser for MacOS X)
- three different e-mail clients (**Netscape Communicator**, **MS Outlook** and **Mail**)
- **iPhoto** (for organising my digital photographs)
- **iMovie** (for editing digital movies)
- **Adobe Photoshop** (for editing digital photographs)
- **Citrix ICA** thin client (allows me to connect to my work computer from home)
- **Toast** (for burning CDs)
- **Print to pdf** (a shareware program for creating pdf files)
- **Adobe Acrobat** and **Preview** (for viewing pdf files)
- **Macromedia Flash** (for developing animated graphics)
- **Home Page** (an ancient but reliable web page editor)
- some **game** programs
- **Symantec Anti-virus** suite.

But that's not all! On each computer that I have used, a program (or group of programs) called the **operating system** must have been running. So I must add the following to my list:
- **Windows 97** (on the ancient laptop I am using to type these notes)
- **Windows XP** (on another laptop)
- **Windows 2000** (on a computer at school)
- **MacOS 8.1** (on my trusty old Mac clone)
- **MacOS X.2** (on my iMac).

Thirdly, a full list would include all the actual **documents**, **files**, **web pages**, **e-mails** and so on, that I had accessed, as these are also software. That would be too long a list, so I'll ignore it here.

Activity

How about you? Make a list of all the software (programs and operating systems) that you have used over the last few days.

The point about all these is this: they didn't grow on trees! They are available for us to use because they have been designed and created by teams of software developers. In this unit, we are going to learn about the process of developing software, and to apply this process to develop some (simple) programs of our own.

Questions

1. What is the meaning of the term **hardware**?

2. Give three examples of **software**.

3. Identify each of the following as either hardware or software:

Item	**hardware**	**software**
monitor		
database		
Windows 97		
scanner		
an e-mail		
Internet Explorer		
mouse		
modem		
a computer game		
a word processor		
digital camera		

2.2 The development process

Before we think about how software is developed, it is worth considering how any product is developed, because the process is essentially the same. For example, think about the process of developing a new model of TV.

Stage 1: Analysis
Before a new product is developed, someone within the company, probably in the marketing department, analyses what people want. They consider which products are selling well, look at what rival companies are producing, and maybe even carry out a survey to find out what people want. From this they can work out which features are required in their newest model, including its size, target price range and various technical requirements.

They use this information to produce a **specification** for the new model of TV. This states clearly all the features that it must have.

Stage 2: Design
The next stage is to turn the specification into a design. Designers will get to work, alone or in groups, to design various aspects of the new TV. What will it look like? How will the controls be laid out? Sketches will be drawn up and checked against the specification. Another team of designers will be planning the internal circuitry, making sure it will allow the TV to do all the things set out in the specification.

Stage 3: Implementation
Once the design phase is over, engineers will get to work to actually build a prototype. Some will build the case according to the design, while others will develop the electronics to go inside. Each part will be tested on its own, then the whole thing will be assembled into a (hopefully) working TV set.

Stage 4: Testing
Before the new model can be put on sale, it will be thoroughly tested. A wide range of tests will be carried out.

It might be tested under '**normal**' conditions. It could be put in a room at normal room temperature, and checked to see that all the controls work correctly, the display is clear, it is nice and stable, and so on.

If it passes this type of testing, it might next be tested under '**extreme**' conditions. For example, does it still work if the temperature is below freezing, or very hot and humid, if it used for long periods of time, or with the volume or the brightness or contrast set to their maximum values.

Finally, it could be tested under '**exceptional**' conditions. What happens if a 2-year old picks up the remote and presses all the buttons at once? What happens if there is a power cut, or a power surge?

If it fails any of these tests, it might be necessary to go back to the implementation (or even design) stage and do some further work, before re-testing.

If it passes all the tests, then the new TV can go into production.

Stage 5: Documentation
However, the development isn't yet complete! Some documentation will be needed to go with the TV – a **User Manual** containing all the instructions about how to work the new TV, and probably a **Technical Manual** for repair engineers.

Stage 6: Evaluation
Once the model is in production, the company will want to evaluate it. Does it do what it is supposed to do? Is it easy to use? And, from the engineer's point of view, is it easy to repair?

Stage 7: Maintenance
Stage 6 should be the end of the story, but in the real world, there needs to be stage 7 – maintenance. There are different kinds of maintenance: fixing faults that turn up once it is being used regularly, improving the design to make it even better, or making changes for other situations (like making a version that will work in another country).

These seven stages are an essential part of the production process.

Activity

OK, let's see if you have got the idea …
Choose any type of manufactured object – it could be a car, an item of clothing, a readymade meal, a toy, a piece of furniture, a building or ….

Now **copy and complete** this table, writing one sentence to describe each of the seven stages in the production of your chosen object:

Object chosen:

Stage	**Description**
1. Analysis	
2. Design	
3. Implementation	
4. Testing	
5. Documentation	
6. Evaluation	
7. Maintenance	

2.3 A dance in the dark every Monday

Exactly the same process goes into the production of a piece of software. The software engineers and their colleagues carry out all the stages of the software development process in order – analysis, design, implementation, testing, documentation, evaluation, maintenance.

Activity

Consider the production of a new game program by a software company.

Here are descriptions of the seven stages, but they are in the wrong order.

Copy and complete another table like the one below, and slot the stages into the correct places:

A. Writing a user guide and technical guide for the software.

B. Deciding what type of game you want to create, and what features you want it to have.

C. Adapting the game to run on a different type of computer.

D. Actually writing all the program code.

E. Checking that the program does what it is supposed to do, is easy to use, and can be fixed if there is a problem.

F. Working out the details of what the screens will look like, what menus and functions there will be, and other detailed aspects of the program.

G. Getting users to try out the program to make sure it works under most conditions.

Stage	**Description**
1. Analysis	
2. Design	
3. Implementation	
4. Testing	
5. Documentation	
6. Evaluation	
7. Maintenance	

Check your answers on the next page.

You should have the following:

Stage	**Description**
1. Analysis	B. Deciding what type of game you want to create, and what features you want it to have.
2. Design	F. Working out the details of what the screens will look like, what menus and functions there will be, and other detailed aspects of the program.
3. Implementation	D. Actually writing all the program code.
4. Testing	G. Getting users to try out the program to make sure it works under most conditions.
5. Documentation	A. Writing a user guide and technical guide for the software.
6. Evaluation	E. Checking that the program does what it is supposed to do, is easy to use, and can be fixed if there is a problem.
7. Maintenance	C. Adapting the game to run on a different type of computer.

In this course, especially from Section 3 onward, you will be putting this **software development process** into practice when you produce some simple programs in a high level computer programming language.

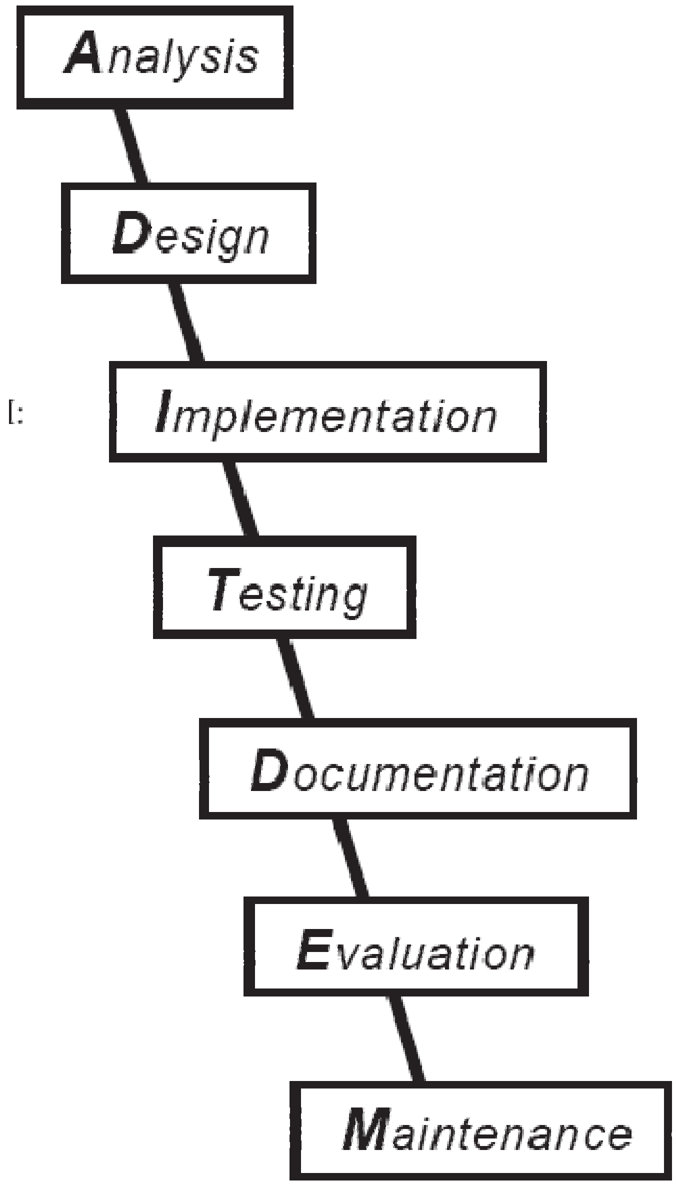
For the moment, it is worth trying to learn the steps in the correct order. I usually use a silly mnemonic for this:

A Dance **I**n **T**he **D**ark **E**very **M**onday

which helps me remember ADITDEM:
Analysis
Design
Implementation
Testing
Documentation
Evaluation
Maintenance.

You might be able to make up a better mnemonic than this one – so long as it helps **you**, then it's OK!

Next, we will take a closer look at each of the stages.



2.4 Analysis

The main purpose of the analysis stage is to be absolutely clear about what the program is supposed to do. Often, a new program will start from a rough idea. Before getting started, it is important to turn the rough idea into an exact description of how the program will behave. What will it do? What are the inputs and the outputs? What type of computer is it to run on? All these questions, and many more, must be asked and answered at this stage.

The result of this is the production of a **program specification**, agreed by both the **customer** (whoever wants the program written) and the **developer** (the person or company who are developing the program).

2.5 Design

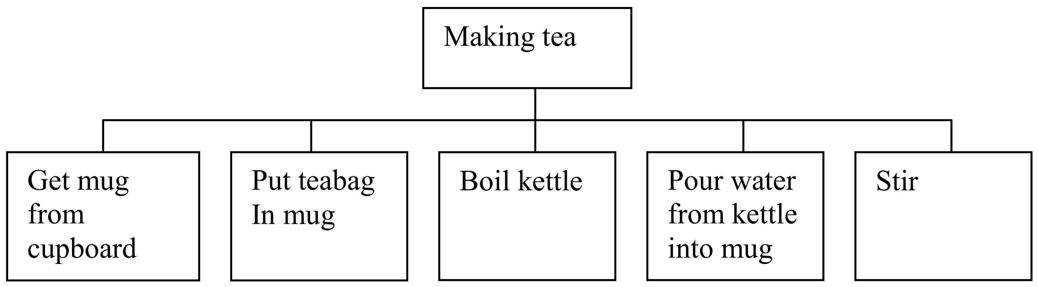
Inexperienced programmers are often tempted to jump straight from the program specification to coding, but this is not a good idea. It is worth spending time at the design stage working out some of the important details, including how the program will look on the screen, how the user will interact with the program, and how the program might be structured. Program designers use a variety of methods for describing the program structure. Two common ones are called **pseudocode** and **structure diagrams**. There are many others, but we will only consider these two.

It is easy to understand these if we think about an everyday example, rather than a computer program.

Think about making tea. Here is a list of instructions for this task:
1. Get a mug out of the cupboard
2. Put a teabag in it
3. Boil the kettle
4. Pour boiling water from the kettle into the mug
5. Stir.

This is an example of **pseudocode**. It is a numbered list of instructions written in normal human language (in this case, English). It doesn't go into all the details, but it gives the main steps.

Another way of showing this is as a **structure diagram**.
It could look like this:



Each instruction goes into a separate box. You read **pseudocode** from top to bottom. You read a **structure diagram** from left to right.

Activity

Now try a couple for yourself. Here are some simple tasks.

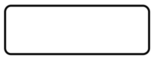
Going to school
Going to New York
Having a shower
Phoning a friend
Becoming a millionaire

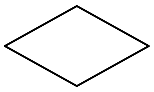
Choose any **two**, and write a pseudocode instruction and draw a structure diagram for each one.

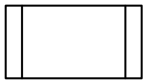
Don't make it too complicated. In the tea example, I broke making tea down into five steps. You could have broken it down into many more detailed steps. For example, getting a mug out of the cupboard could be broken down into smaller steps – walk across to the cupboard, open the door, choose a mug, lift it out, close the door, walk back across the room. Try to break the task down into between four and eight steps.

We will use pseudocode in Section 3 when we start to develop our own computer programs.

There are other graphical methods of representing the structure of a program. These include **structure charts** and **flowcharts**. Some use a variety of ‘boxes’ to represent different types of instruction. For example, you might see:

 to represent a **repeated action**

 to represent a **choice**



to represent a step which will be **broken down into smaller steps**

2.6 Implementation

In software development, implementation is the process of converting a program design into a suitable programming language.

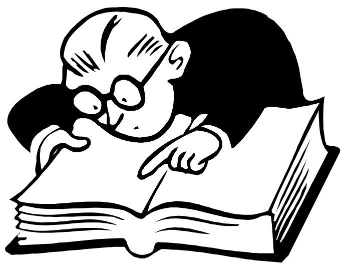
There are thousands of different programming languages out there, all with their own advantages and disadvantages. For the purposes of this course, you only need to know about two main groups: **machine code** and **high level languages**. You will learn more about these in Section 2.

2.7 Testing

We looked at testing at the start of this section. Whether we are talking about a new TV, a new item of clothing, or a new computer program, the manufacturers will spend a great deal of time on testing. This will be carefully planned to test a wide range of conditions. We can divide it up into three types of testing.

- **Testing normal conditions**
  Making sure the program does what it should do when used ‘normally’.
- **Testing extreme conditions**
  Making sure the program can handle situations that are at the edge of what would be considered normal.
- **Testing exceptional conditions**
  Making sure it can handle situations or inputs that it has not been designed to cope with. You will see examples of all of these in Section 3.

2.8 Documentation



When you buy a product, whether it is a computer program or anything else, you usually get some kind of **User Guide** with it. This tells you how to use the product. It might also contain a tutorial, taking you through the use of the product step by step.

Some software comes with a big fat book called User Guide or Manual; others come with the User Guide on a CD.

As well as documentation for the user of the software, there should also be a **Technical Guide** of some sort. This gives technical information which is of little interest to most users, except that it will usually include information about the specification of computer required, including how much RAM it needs, how fast a processor it must have, and which operating system is required.

The Technical Guide should also include instructions on how to install the software.

Activity
Get hold of a software package that has been bought by your school or college, or one you have bought yourself at home, open it up and take a look inside the box that it came in. Make a list of all the items of documentation that you find there.

2.9 Evaluation
The final stage in the process before the software can be distributed or sold is evaluation. Evaluation involves reviewing the software under various headings to see if it is of the quality required.
In this course, we will review software under three headings: **fitness for purpose, user interface** and **readability.**

Is the software **fit for purpose**? The answer is 'yes' if the software does all the things that it is supposed to do, under all reasonable conditions. This means going back to the program specification (produced at the analysis stage) and checking that all the features of the software have been implemented. It also means considering the results of testing, and making sure that the program works correctly and is free from bugs.

The **user interface** should also be evaluated. Is the program easy to use? Is it clear what all the menus, commands and options are supposed to do? Could it be improved in any way?

The third aspect of evaluation that we will consider is **readability**. This is of no direct concern to the **user** of the software, but is important for any **programmer** who may need to understand how the program works.

It is to do with the way that the coding has been implemented. Is it possible for the program code to be read and understood by another programmer, perhaps at a later date when the program is being updated in some way? We will look in Section 3 at some techniques for improving the readability of a program.

2.10 Maintenance

This final phase happens **after** the program has been put into use. There are different types of maintenance that might be required. These are called corrective maintenance, perfective maintenance and adaptive maintenance. You don't need to know these names until Higher level, but it is useful to think about what they mean.

Corrective maintenance means fixing any bugs that appear once the program is in use. Of course, these should all have been discovered during testing. However, most programs (but not the ones you will be writing) are so huge and complex that some bugs are bound to slip through unnoticed. If the bugs are serious in nature, the software company might issue a free 'patch' on its website, so that users can download the patch, and install it with the software, so fixing the bug. If it is a minor bug, they may not bother.

Perfective maintenance is adding new features to the software. These might be suggested as a result of the evaluation stage, or they might be suggested by users. These new features will then be added to the software, and re-issued as a new version. That's why software often has version numbers. Each version results from corrective and perfective maintenance of the earlier versions. So (for example), BloggProg 3.2 will be similar to BloggProg 3.1, but with bugs fixed, and some new features added.

The third type of maintenance is **adaptive maintenance**. This is where the software has to be changed to take account of new conditions. The most obvious example is when a new operating system comes out. Perhaps BloggProg 3.2 was designed to run under Windows 2000. When Windows XP came along, changes had to be made to BloggProg so that it would work under the new operating system.

Questions

1. Match up these descriptions of the stages of the software development process with the correct names (one has been done for you):

Stage	Description
Evaluation	Writing a user guide and technical guide for the software
Testing	Working out the details of what the screens will look like, what menus and functions there will be, and other detailed aspects of the program.
Implementation	Deciding what type of game you want to create, and what features you want it to have.
Design	Actually writing all the program code.
Documentation	Adapting the game to run on a different type of computer.
Analysis	Checking that the program does what it is supposed to do, is easy to use, and can be fixed if there is a problem.
Maintenance	Getting users to try out the program to make sure it works under most conditions.

2. What three criteria will be used for evaluating software in this unit?
3. What is the relationship between pseudocode and a structure diagram?
4. Name two items of documentation usually provided with a software package, and describe what you would expect each one to contain.
5. What three types of testing should be applied to any software?
6. Describe two examples of maintenance that could be required on a game program.