# Computing

## Software Development (Int 2)

August 1998

HIGHER STILL

# Computing

## Software Development
## Intermediate 2

Support Materials

**TEACHING AND LEARNING PACK CONTENTS**

## SECTION 1 TEACHER/LECTURER NOTES

### Aim

This unit is designed to develop knowledge and understanding of the software development process and of a software development environment. It provides the opportunity to implement features of a selected software development environment, to use its facilities and to produce a software solution to a problem.

### Status of this teaching and learning pack

These materials are for guidance only. For the specification of the content of this unit you should refer to the Arrangements document.

### Target audience

While entry is at the discretion of the centre, students would normally be expected to have attained one of the following (or equivalent experience):

- Software Development (Int 1) unit
- Computing Studies Course at Intermediate 1 level
- grade 3 or 4 at Standard Grade in Computing Studies.

### Progression

Students should have basic knowledge and skills in computing before tackling this unit. However, no prior knowledge of software development is assumed in these materials as Software Development at Intermediate 1 is an optional unit.

Students are expected to have completed Software Development Int 2 before attempting the Project (Intermediate 2). Students may choose to deepen their knowledge and extend their skills in a subject area such as software development in the Project.

Software Development is a mandatory unit in Computing Higher. The recommended entry to Software Development (Higher) is one of:

- Software Development (Intermediate 2)
- Computing Course (Intermediate 2)
- grade 1 or 2 at Standard Grade in Computing Studies.

### Hardware and software requirements

Students can use any software environment whose features and facilities can be used to produce solutions to problems. An integrated environment with a 'programming' language and facilities such as a text editor and debuggers would be ideal. An imperative language, a declarative language, an applications package with a high functional capability, an object-oriented language, a scripting language or an authoring environment are all suitable. The support notes are exemplified in a procedural high level language. At this level knowledge and skills should be developed using one software environment. The hardware will be a platform which supports this environment. Each student needs about 25 hours access to a system during the unit.

**Learning and teaching approaches**

The theoretical aspects of this unit should be introduced in the process of developing skills in analysis and design, implementation and testing, and evaluation. However it is recommended that when students have developed some skills and understanding in software development, they are given time to review and discuss the various aspects so that they can assimilate the relevant terms and use them appropriately.

**Pathway through the unit**

The structure of this pack follows the order of the unit outcomes. However, it is more likely that the student will learn about software development through learning about:

- analysis and design
- control structures within a particular programming environment
- the importance of both of these aspects within the software development process.

In other words, the student is likely to tackle the outcomes of this unit in reverse order.

The notes and tasks for Outcome 3 take the student from a simple problem to the point where they can solve problems at the level of the assessment for this unit. The tasks set for the student are very similar to the preceding exemplar used to introduce a particular teaching point. These exemplars are written in Pascal and teachers/lecturers wishing to use a different programming environment may wish to translate the exemplars into their preferred language. Teachers/lecturers may want to supplement these tasks with some of their own.

At some point in the unit, probably in the second half of the unit, the student will need to have an overview of the software development process and its constituent elements. At that point, the student may use the study materials for Outcome 2 and then Outcome 1 as a basis for discussion and assimilation of the software development process as a whole.

The sections may form a useful review before students tackle the problem in Outcome 4 and the assessments for the unit. They presume that students have heard and are beginning to understand the following terms:

**Software development**

iteration
problem specification
robust
correct
valid
development cycle, including maintenance
algorithm
top-down design, stepwise refinement
control structures - sequence, repetition, selection
user interface

**Hardware related**

input/output devices
backing store

The materials also presume that the students can read simple algorithms.

## SECTION 2 STUDENT'S NOTES

### What will I learn

In this unit you will learn how to solve problems using a programming language.

Firstly, you will learn how to analyse problems and break the solution down so that the stages in the solution are clear (Outcome 3).

You will have to learn use a programming language so that you can code the solutions. This means that you will learn the important words in the language and how to use them (Outcome 2).

Through this you will learn about the different stages in the software development process - analysis, design, implementation, testing and evaluation. You will be able to both do and describe these stages (Outcome 1).

Finally, you will put all this into practice by developing a solution to a problem which is similar to those that you will have to solve for the unit assessment (Outcome 4).

Do not worry if this sounds as if you are doing the outcomes in the wrong order. Your teacher/lecturer will make this clear and easy for you to learn.

### How will I be assessed?

In the assessments you will be asked to:

- describe and recognise the different stages of the software development process
- describe and use key aspects of your programming language
- solve problems similar to the last few done as part of your learning in the unit.

## SECTION 3 STUDY MATERIALS

## SOFTWARE DEVELOPMENT PROCESS

## OUTCOME 1

### Stages and iterative nature of programming

Iteration means to repeat certain steps.

For example:
Set a total to zero
*Repeat*
- Think of a number less than 10
- Add it to the total

*Until* The total is greater than 100.

This is an iterative process. You can see that the steps are followed in order and many steps are repeated.

This is true of the development of software. It is an iterative process.

Many steps are involved - analysis, design, implementation, testing, documentation, evaluation and maintenance. The stages of the development process are repeated in debugging a program, adapting it to suit a new situation.

Just look at any software packages on the shelves of the large computer stores. When new software appears in the stores it is generally labelled version 1.00. As you scan the shelves you may notice a version number, perhaps version 3.12. The '3' indicates the second major revision of the software. The '1' is a smaller alteration. The '2' is yet smaller.

A wide range of skills are needed to write a program to solve a problem.

We will now look in detail at the various stages of software development mentioned above - namely: analysis, design, implementation, testing, documentation, evaluation and maintenance.

### *Analysis of the problem*

This is the process of writing out clearly what the solution to a problem must do.

What type of computer and what operating system will be used to run the program?
Will it need special input or output devices - graphics tablets or plotters?
Will it be using specialist backing storage devices like CD-ROM, magnetic tape?

### *Problem specification*

A professional programmer may only have a rough outline of the problem. This must be changed into a clear and precise description of the problem - the *problem specification*. What is involved must be clearly understood by both the programmer and the client. It must be agreed by both before the design of the program begins.

Think of large banks. If new software was being developed for all banks across the country the cost would be very high. It would be important to get the new software correct first time to save money and keep good customer relations. This would involve many analysts speaking to senior management, local management and customers to ensure that they knew what the software must do.

Well designed software which is robust and easy to use is the aim. Program schedules (the expected time for each stage of development), test plans (to check that it is robust and correct) and user documentation (instructions on how to use the package) arise from the problem specification.

### *Implications of errors in the problem specification*

Errors in the problem specification should be detected early in the development cycle as they are costly to correct at a later stage. If this is not done poor testing and incorrect documentation will follow. It costs far more to correct an error at the maintenance stage of the software cycle than it does at the analysis stage. An essential starting point therefore is a formal specification of the problem.

### *Algorithm*

An algorithm is a set of instructions that describes how a particular type of problem may be solved. Algorithms have been around for a lot longer than computers.

Suppose you are going to make pancakes. Firstly you will get the ingredients - in this case, flour, sugar, eggs and milk. These will be combined together following the instructions of the recipe. When the cooking stage is complete you will have a plate of pancakes. Anytime you make pancakes the same recipe instructions will be followed.

Taking the car out of the garage is another example of an algorithm. The steps in this case are: unlock the garage door, unlock the car, get into the car, put the key in the ignition, turn on the engine, select reverse gear, put off the handbrake, check that there is no obstruction, reverse car out of the garage.

Think of the pancake recipe again. We will only get pancakes if we use the suggested ingredients. If instead of flour, sugar, eggs and milk, we used eggs and milk only we would end up with an omelette.

You can see from this example that we must clearly identify the input to a recipe to obtain the output that we expect.

In the example of taking the car out of the garage certain assumptions have been made. We are assuming that the car is (1) locked (2) not alarmed (3) in neutral gear.

We are also assuming that the garage door has not six feet of snow in front of it making it impossible to open easily.

Let us go back to the pancake recipe again. Compare the situation of making pancakes for supper for four people with making pancakes for a youth club party. Obviously far more pancakes are needed for the party. However the same recipe will be used - that is the same algorithm. The inputs (ingredients) will be altered which will affect the outputs.

An algorithm, when correctly implemented will always produce a correct result. Hence the correctness of a program in computing can be ensured if the correct algorithm is chosen.

We can see from the above examples that a problem is only fully specified when it is stated clearly and we define the inputs and outputs for the problem. Moreover we should state any assumptions that we are making.

There may be several ways of tackling a problem but the most efficient algorithm should be selected.

Here are some classic and simple algorithms commonly used in computer programming:

1.The Counter

This is used to count how many times you do something. At the start the counter is set to zero and every time something is done 1 is added to the counter.

```
1. counter = 0                      start count at zero
2. begin a loop                     something is going to be done more than once
3.         do something
4.         counter = counter + 1    add one to your count
5. end loop
6. display counter                  you can see the final value of counter
```

In the example above you will see a segment of six lines of a program design. The instructions on lines 3 and 4 are indented. These are the instructions which are within the loop and will be repeated a number of times.

2. Validation

If the entry at the keyboard is wrong the user should be asked to reenter the data. Suppose the program asks you to enter a number between 1 and 100. You enter 141. This should be rejected and you should get a polite message to enter the number again.

| | |
|---|---|
| 1. get a value | |
| 2. while value is wrong | it could be correct! |
| 3. display an error message | user warning |
| 4. get a value | will it be right this time?? |
| 5. end while | right at last - we go on to the next part of the program |

In this example the segment has five lines of a program design. The instructions on lines 3 and 4 are indented. These are the instructions which are within the while - endwhile loop and will be repeated until a condition is satisfied.

**Questions on Outcome 1.(a)**

1. What are the stages involved in the development of a program?
2. When developing a program what must be clearly understood and agreed between the client and the programmer?
3. What is a program specification?
4. When in the development cycle is the best time to identify errors?
5. What is the effect of errors not being identified before a program is in use?
6. A student tries out a new program called 'Hercules'. He notices that it is labelled version 4.3. Explain the meaning of the expression version 4.3.
7. The software development cycle is described as an iterative process. Explain what this means.

The answers to these questions are at the end of Outcome 1.

***Design methods***

When the problem for the programmers is clearly specified there are some techniques which the programmer can use to produce the program. Having decided on the algorithm a method of representing the design will be selected. A number of methods is available. What these all have in common is the means of expressing a large problem as a set of smaller problems, etc. This method is called Top Down Design. The process of breaking a problem down into smaller problems is known as stepwise refinement. Stepwise refinement continues until the subproblems are manageable enough to be coded.

These techniques can be used outside computing. The design for a book could look something like this:

1. Beginning

    1.1 set scene
    1.2 introduce main characters
    1.3 introduce plot

2. Middle

    2.1 develop relationship between main characters
        2.1.1introduce a minor but important character
    2.2 develop plot
        2.2.1 tell strand 1 of plot
        2.2.2 tell strand 2 of plot
        2.2.3 tell strand 3 of plot

3. End

    3.1 bring together all strands of plot
    3.2 conclusion of plot
    3.3 happy end.

Two graphical methods of designing the structure of a program which we will look at are structured diagrams and flow charts. We will also look at a method which describes the design of the program in English text style.

**Graphical representation of the program design - structured diagrams**

The diagrams produced are called structured diagrams. They represent the top down design of the solution by 'layering'. The top layer of the diagram is the initial problem. The second top layer shows the breaking down of the initial problem into smaller problems. It also shows horizontally from left to right the order in which the smaller problems will be addressed. The third layer shows the further breakdown of the smaller problems etc.

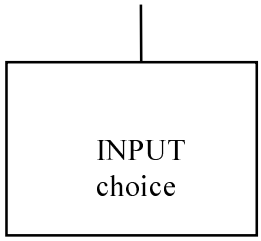
Some of the structures and their meanings are shown on the next page.

**SINGLE ACTION symbol**

This represents a single processing action

e.g. INPUT choice

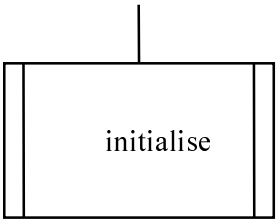
There is no exit from the bottom of the box




**PROCEDURE symbol**

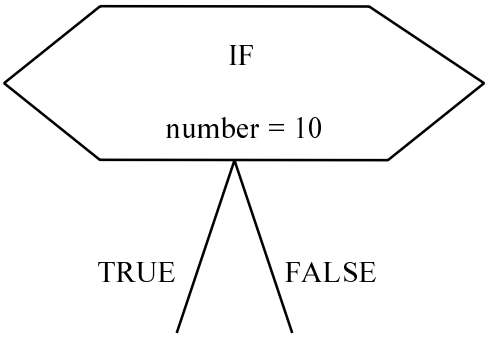
This represents a single process

e.g. initialise




**DECISION symbol**

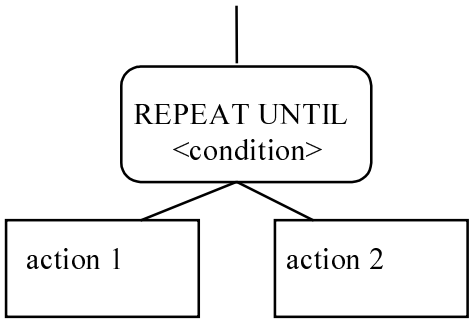
A decision is made and only one exit path is taken




**Structure Diagram Symbols**

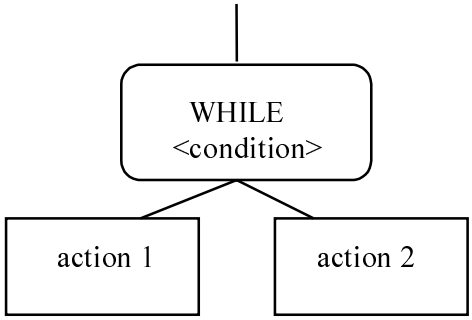
**REPEAT loop**

The actions to be performed are listed below the loop



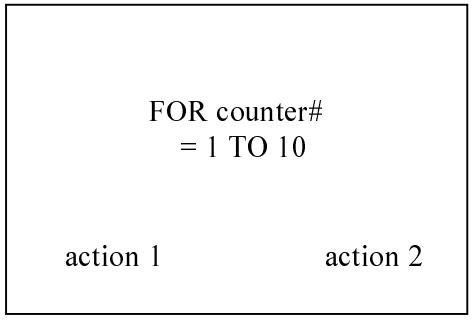

**WHILE loop**

The actions to be performed are listed below the loop




**FOR counter# =**
**1 to 10 loop**

The actions to be performed are listed below the loop




**Repetition Structures in Comal**

**In each of the three loops above only two actions are shown.**
**In fact any number of actions are possible**

The design for a program in which the computer will simulate a coin tossing game is shown on the next page.

The user will be asked to predict whether the outcome will be heads or tails ten times.

The program will simulate a coin toss after each response and display the result on screen with an appropriate message indicating whether or not the prediction was correct.

A running total of correct responses will be kept and displayed as a percentage of the ten guesses at the end of the program run.

**Assumptions**

1. The user will strike a key (‘1’) on the keyboard to indicate a ‘head’ or (‘2’) to indicate a ‘tail’ in the toss.

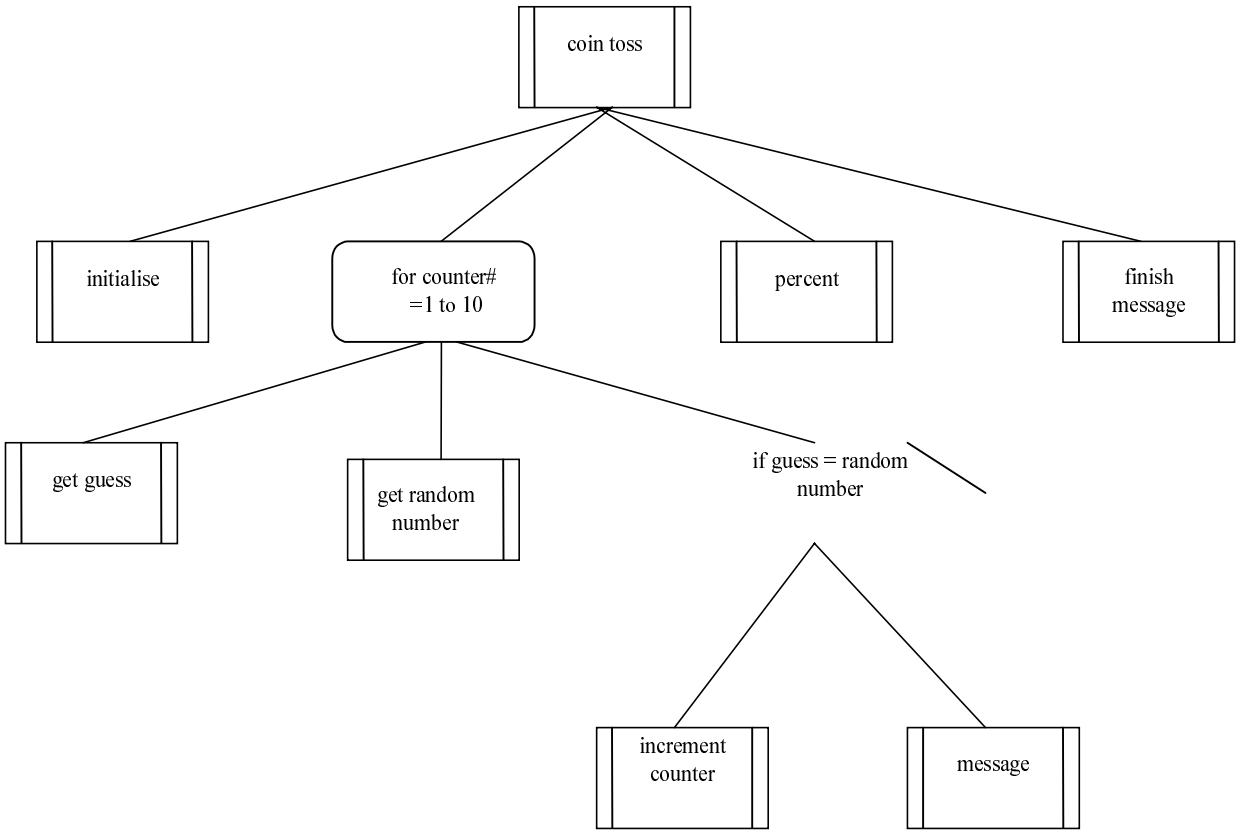
2. The percentage will be rounded to 1 decimal place.

Complete structured diagrams for the procedures on the third level of the diagram.

**Graphical representation of the program design - flowcharts**

Program design can be shown in flowcharts. Look at the following example.

Take in nine numbers from the keyboard. Each one should be added to the previous. The total of the nine numbers should be displayed.



**Representation of the program design - pseudocode**

Pseudocode is writing the program design in English statements using key words. Stepwise refinement can be identified by the numbering of the statements. Indentation of the statements helps to represent the design structure. This helps to make the program more readable.

***Example***

Write the Top Level Design for a program to read in the length and breadth of a rectangle and display the area.

Top Level Design

1. set screen output mode
2. description of program
3. get length
4. get breadth
5. find area
6. display area

Refinement of subproblem 3

3.1 ask for length
3.2 enter length

Refinement of subproblem 4

4.1 ask for breadth
4.2 enter breadth

Refinement of subproblem 5

5.1 area = length multiplied by breadth

**Choice of design technique**

You have seen that the design of a program can be represented graphically and in English statements. Three examples have been used to illustrate this. However generally programmers use only one, or possibly two methods. The teacher or lecturer will probably indicate which method he/she prefers.

***Problem***

Using the method advised at your centre or school, design a solution for the following problem.

Ten pupils take part in a sponsored walk of 13 miles. They want to use the computer to calculate how much money is raised by each and the total for all ten. The names of the 10 pupils are to be entered at the keyboard along with the amount each is sponsored. The name, total sponsor money for each and the total for the group should be displayed.

## Implementation

### *Selecting a programming language*

When the design of the program is complete and has been thoroughly tested we will need a high level programming language to code it. There are many high level languages available, Comal, Visual Basic, C++, Pascal, Basic etc.

Have you used an application package like Clarisworks ? Do you remember how easily you could develop the solution to a problem with it? You were helped by the 'friendliness' of the screen messages. It had on-line help. If you did not know how to do something, with the click of a button you had help instructions on the screen. You did not have to remember commands. The menus presented the options on the screen. You could select your choice by picking and pointing with the mouse. You did not need to know the rules of the language. The program helped you to learn.

The best way to learn a language is to use it - but for High Level Programming Languages you must know the vocabulary and the rules. You must know what control structures are available in the language. These languages take more time to learn than application packages like Clarisworks.

*The following questions will help you to choose a programming language.*

How complex is the problem?

Can it be solved by using a software application? If not …

Which language will be the most effective for the particular problem?

What hardware is needed for the language?

What programming language do I know?

Can I include numbers, letters and other characters such as - a question mark, a percentage sign? What about video, sound and graphics?

Can I develop a program in modules, that is, with procedures?

Is there a library file of procedures? Can I add any I develop myself to the module library file?

Does the language let me do structured programming? Does it support sequence, repetition and selection?

Does the language let me enter data into the computer and also get results out?
Pascal is a high level language developed by Professor Niklaus Wirth of Switzerland. It encourages the use of structured programming. This may be your choice.

A High Level Language means the instructions you write in the source code are all English-like words which are more meaningful to you or anyone else who may use your program.

Once you have coded the design of your program in a High Level Language you will run the program on a computer which has a translator program for the code.

All programs developed in High Level Languages have to be translated into machine code - the only language the computer 'understands'. The method of translation will depend on the High Level Language you have chosen. The translator program may be an interpreter or a compiler depending on your choice of High Level Language.

When the coding of your program in Pascal is ready to be entered into the computer, you will use a text editing package. The file you create will be called the source code. It will be processed by a special program called a compiler. First the text file will be checked for syntax errors (have you broken any rules of the language?) The program construction will be checked.

In general good programming style will be friendly to the user; be as short as possible at the top level; have clear internal documentation; use meaningful variable names; have routines to check that all inputs are acceptable; and show a clear structured layout.

If your text file (source code) is free from errors then the compiler will translate the words into machine code. This Low Level Language file of code is called the object file. When you want to run your program this object file is loaded into the computer's memory and executed.

The Pascal Compiler must know the type of any variables which are used as source code. This means that variables must be identified as integer, real, character or string.

**High Level Language features**

***Reserved words***

The language you use to program will have reserved words. These are words which have a special meaning in the High Level Language. They cannot be used as names for variables.

The following list shows some examples of reserved words in Pascal, Comal and Basic.

| Pascal | Comal | Basic |
|---|---|---|
| AND | CLS | END |
| DOWNTO | PRINT | PRINT |
| IF | MODE | REPEAT |
| ELSE | END | WHILE |
| REPEAT | INPUT | |
| WHILE | REPEAT | |
| UNTIL | GET | |
| END | | |
| PROCEDURE | | |
| FUNCTION | | |

A computer program is not very useful if it cannot be run many times using different data. To make programs more adaptable variables are used instead of values. A program will be more readable if variable names are meaningful. 'entry1' and 'entry2' for example are not very meaningful. You will write a lot of programs. Will you remember yourself when you look at a program several months later what the 'entry1' and 'entry2' stood for? If you were storing a name you are more likely to remember it if you called it 'name$'.

So how do we give a name to a variable?

***Assignment statements***

We use an assignment statement. Examples:

(1) counter := 10.8 - a variable called counter is now identified in the computer's memory and has the value 10.8 stored. If we followed the assignment statement with a print statement then 10.8 would be written out.
(2) longstring$ := onomatopeia - a variable called longstring$ is now identified as holding the data onomatopeia
(3) primenumber# := 31 - a variable primenumber# is holding the value 31.

The three variables in the examples are similar because they contain words which are meaningful in each context. However, on closer inspection you see that a # and $ symbol are used. # identifies an integer and $ represents a string that is, a group of letters. longstring$ is a string variable.

***Operations***

We want to be able to perform operations on numbers like we do in arithmetic. We need a means of representing these operators.

The use of symbols like + (add) - (subtract) * (multiply) / (divide) will let arithmetic operations be done.

High level languages will help you to develop your program by offering you structured programming.

Structured programming is based on three main techniques:

**SEQUENCE** *the program does one thing after another*

1. get firstnumber
2. get secondnumber
3. answer is firstnumber plus secondnumber
4. write answer
5. end

**REPETITION** *an action is repeated until a condition is met*

known number of repeats

1. for count from 1 to 5
2. write 'happy birthday'
3. next count

repeat continues until a condition is met
test is at start of loop

1. get a word
2. while word is not 'lucky' *test condition*
3. get a word
4. endwhile

repeat continues until a condition is met
test is at end of loop

1. repeat
2. get a number
3. until number is less than zero *test condition*

**SELECTION**

1. if number is odd then
2. display message1
3. else
1. display message2
2. endif

Sequence means the program will do one thing after another.

Repetition means the program repeats an action either a fixed number of times or until a condition is met.

Selection allows a program to choose a course of action.

Most structured programming languages then allow you to develop the solution to a problem in a modular way, that is, they will let you use procedures. Remember - a module is the same as a procedure. Frequently when you have refined your program into modules you will see a similarity between one of these modules and one that you have coded before. It is a good idea if you can save such module designs in a library file. This will save you designing them again. Functions are also coded for you. They have been thoroughly tested and are regarded as robust and reliable. This saves you writing them.

To sum up then structured programming languages allow you to use control structures for selection, unconditional, simple conditional and sequence statements. They provide you with a means of representing different types of data objects. They let you perform arithmetic operations. They provide structured listings of the programs.

***Structured listing***

Apart from letting you design a program by Top Down Design, another help that a structured programming language offers you is structured listing. Indentations in the lines of programs make the program easier to read. They also helps you to see where the beginning and end words in structures match up.

Example 1

```
        get a number

IF      number is greater than 25    THEN
        write 'well done'

ELSE
        write 'better next time'

END IF
```

***Testing***

It is important that any software solution developed for a problem should produce correct results. The only way this can be checked is to run the program and test it with a variety of inputs and see if the results produced are what you would expect.

For the example above, try testing with the following: 34, 12, 0, 78, 25 and check for yourselves that the results you expect are produced.

Example 2

Below is some pseudocode with a simple conditional

```
        get a number
```

```
        square it
        add the number to its square
IF      answer is greater 100          THEN
        write 'accepted'
END IF
```

***Test data***

A program cannot usually be tested for every possible input.

Apart from structured listings another technique which will help you to develop correct, reliable and robust software is careful selection of test data.

Example 1

Break the following problem into smaller problems.
Take two numbers, add them together. What is the answer?
This problem naturally divides into three smaller problems.

1. get numbers
2. add numbers together
3. display results

These smaller problems are called modules or procedures. Ideally a module should only tackle one task. This means each module will be easy to test. The above example is simple and it would be easy to choose test data. There is just one path through the design. Only a sequence of statements is represented.

Example 2

This problem is about grades in an exam. If the student scores 75 or more the grade is '1'; between 55 and 74 the grade is '2';otherwise the grade is '7' The marks range from 0 to 100. This problem has selection constructs.

Top Level Design

```
get mark
IF              mark >= 75          THEN
                grade '1'
ELSE
                IF                  55 <=mark <=74    THEN
                                    grade '2'
                ELSE
                                    grade '7'
                ENDIF
ENDIF
display grade
```

We choose test data which will be valid for each grade. Also we test the limits of the grades and finally we use some data which will test for unacceptable inputs.

Try this selection: 83, 69, 45, 75, 74, 55, 54, 100, 0, 132, -56.

***Error reporting***

After a problem has been broken down by top down design and stepwise refinement each module should then be tested separately. The software will have a text editing facility. The modules have to share data with each other so they should be tested to see how they will work together. Some programming languages will provide automatic help to the programmer by emboldening key words. Indentation in constructs may also be provided. Completion of constructs may be done - like FOR…NEXT loops in Comal.

Another help which the programming software offers is interactive debugging. This may include tracking a variable through a program and noting the values it takes. Breakpoints can be set to stop the program executing at a point. The contents of all variables which have been assigned values up to the breakpoint will be kept and can be checked by the programmer. Step commands can also be used. The program will step through a statement at a time. Thus each statement can be checked for error.

Error reporting facilities in a software development environment help to increase the rate of production of reliable and robust code because the programmer is alerted to any errors immediately.

Thorough testing and debugging are essential. The process of debugging a program involves looking for *syntax errors*. These arise when the rules of the language are broken. *Run-time errors* arise during execution when the program attempts for example to divide by zero or find the square root of -2. *Logical errors* are found when the syntax is correct and the code runs without any error. However the program does not give the correct result. If you wanted to find the square root of 81 you would expect the answer to be 9 or -9. The program output is 6561!! You have squared the input rather than taking the square root. This type of error is difficult to find because the program will compile all right. It just is not giving a correct result.

The correctness and reliability of the program can only be determined by very thorough testing after debugging.

Think again of this problem. Take two numbers, add them together. What is the answer?

How could we test subproblem 1 above?

We could test each input to see that it was a number - not a letter, character word, etc.

The output from subproblem 1 must go into subproblem 2. The arithmetic operation - ADD - only works on numbers. The input to subproblem 2 must be a number. Hence the subproblem 1 *get numbers* would have to call on another module which would test that the inputs were indeed numbers.

Once a module is tested and any bugs removed it may be entered into a module library. It could then be used in other programs. The module to test for acceptable input described above will be used many times so the advantage of having a library of modules can be seen. Time has been spent developing the module so storing it will make it always available whenever required. Another advantage of a module is that it could be called anywhere in a program - any number of times. All these features of a module save time in the program development.

***Documentation***

Documentation is needed so that users of the program know how to use it. The user guide should make clear to the user all that he/she needs to know; the start-up instructions for the program, the facilities it offers and how to use them. A technical guide should be provided. This gives instructions on how to install the software. It details hardware and memory requirements. Documentation is essential if any person other than the programmer is going to use the program. Look at the documentation which has come with some application packages like word processing with which you are familiar.

At all stages of software development documentation should be evident.

| **Stage** | **Documentation** |
|---|---|
| analysis | problem specification and program description, inputs and outputs, assumptions |
| design | flowcharts/structured diagrams/pseudocode |
| implementation | internal commentary |
| implementation | program and output listings |
| implementation | test history and sample outputs |

***Evaluation***

To evaluate our solution we must go back through each stage of the development of the software and check whether the solution meets the requirements. In analysing the problem have we correctly interpreted it? During design have we taken account of user interaction with the program - helpful screens, internal documentation and modularity. Is the program reliable? robust? Can it be easily maintained? Is it giving correct results? Is this the best solution? Has a user guide been included? Is there a technical guide? Have potential changes been suggested?

***Maintenance***

As time passes programs may have to be altered for a variety of reasons.

The working practices of the customer may have changed and this affects the format of the data being used as input.

The program may have to be adapted to handle and process more files if a client's business expands.

An error not detected in the testing of the program could show up.

A rival software house may have produced a new version of similar software which includes more features.

Developments in operating systems and processors mean that the way the program interacts with the computer will have to be changed.

Can subsequent changes be effected easily and quickly?

This will be determined by the answers to the following questions.

Have meaningful names been used for variables? Is there internal documentation which is helpful? Is there a user guide? Is there a technical guide? These all aid the maintenance of a program.

In general good programming style will be friendly to the user; be as short as possible at the top level; have clear internal documentation; use meaningful variable names; have validation testing routines and show a clear structured layout.

**Questions for Outcome 1(b)**

1. While developing the program what help is available to the user?
2. What does the expression 'Top Down Design' mean?
3. What is the advantage of this design technique in software development?
4. What is meant by 'stepwise refinement'?
5. Why is it useful to store modules in a library file?

The answers to these questions are at the end of Outcome 1.

## Characteristics of software

At each stage of the design process the programmers must consider certain characteristics of the program they are developing. Some relate to the requirements of the customer who is going to use the software. Some relate to other programmers or computing specialists who may have to read the program.

### *Fitness for purpose*

A program should be:

*correct* A program should produce the correct output for all valid inputs

*reliable* Are there any design or coding bugs? Is the program design correct? For all valid inputs is the expected output given? The test data should be selected carefully - acceptable data, unacceptable data, data at the extremes of a range etc. A dry run may be carried out. This means choosing data and noting down the values it takes at various stages in the program. A trace table may be constructed.

*efficient* The program should run at a speed that is appropriate to the task it has to perform.
There is no point in using a bank auto-teller that takes four hours to complete a transaction.
Likewise a small business payroll program which had to run on a super-computer with Gigabytes of Ram would not find many customers.

### *Design of the user interface*

The screen presentation should be clear. The user should receive consistent instructions on how to proceed through a program. If at different stages the program stops running until the user presses the <space> bar then the screen instructions on how to proceed should be positioned at the same place on the screen. The user will look to that place. Try this yourself. Clear prompts should be given.

The software should be robust. The user should see any data entered on the screen so that it can be altered if necessary. Suppose you wanted to type in '37' but you typed '73' instead. How do you know you have made a mistake in data entry if you do not see what you have typed. It should be clear to you how to amend what you have typed if it is wrong.

All inputs should be validated and helpful messages given to the user. This means that any entry at the keyboard should automatically be passed into a module which will test the validity of the input before accepting it. If your entry is being rejected as invalid - suppose you had entered 31st February 1999 - you should get an error message on-screen.

If invalid data is allowed into the program it may cause the program to 'hang' - this is when a data item is entered which is not acceptable input. The program cannot proceed and you will have no option but to come out of the program.

The program should give correct results for all acceptable inputs. Errors like division by zero should be anticipated and trapped.

The program should present the user with prompts that make sense and are easy to follow.

Menus should have suitable headings. They should be well structured. The option names should be meaningful.

Any error messages should attract the user's attention. Where possible the action which caused the error should be indicated. A way of correcting the fault should be suggested.

Other programmers should be able to look at the source code and understand how the program works.

Meaningful variable, procedure and function names and internal commentary all help to make a program easier to read and debug.

***Documentation***

Your program should include comment lines. These could describe what different parts of the program do.

A user guide describes how to use the program. It describes all the facilities the package offers and how to use them. Frequently nowadays a tutorial, usually in electronic form, will be provided - an on-line tutorial. The user guide should be updated whenever revisions to the program take place. The content should be precisely stated.

A technical guide gives instructions on how to install the software. It details the memory needed to run the program. The type of processor needed is listed. The version number of the software is noted. If any additional help or system files should be installed the user will be informed.

***Maintainability***

Modular design makes a program more maintainable. Any segment of the program which needs corrected, updated or changed can be quickly identified and altered. This is like replacing a faulty component in a computer. The broken part is discarded. If the replacement is of the right type and works properly the whole system will start up and the user should not notice any difference except an improvement in performance.

A program will be more readable if internal documentation has been included. The use of meaningful variable names helps too.

The programming language in which the program is written determines the portability. A suitable translator has to be available for the platform on which it will run. Can the program be adapted to run on other computers? The use of machine specific terms is avoided where possible. When their use cannot be avoided they are 'flagged' by a remark or explanation. This means they can easily be changed to suit a new platform.

**Questions for Outcome 1(c)**

1. A customer wants to withdraw £50 from an ATM dispenser. He follows the instructions on the screen. When he collects his cash he finds that he has £200. Give some suggestions about how this could have happened. In what way could the user interface be altered to avoid a recurrence of this?
2. Another customer uses the ATM. When his money is delivered he takes his card but forgets to lift his money. In what way could the user interface be changed to alert the customer?
3. Describe what is meant by a *reliable program.*
4. A student is writing a program which will print a luggage label. He decides that he will include four pieces of information on the label - his name (he calls this 'item1'); his address (he calls this 'item2'); his destination (he calls this 'item3'); his flight number (he calls this 'item4'). What do you think of his choice of variable names? Suggest alternative names.

The answers to these questions are at the end of Outcome 1.

We will now apply what we know about software development to a problem.

**1. Problem**

Accept two numbers from the keyboard and find the difference between them. The problem specification includes a statement of the problem with details of any assumptions that you are making. Inputs and outputs for the problem should be noted. Top level design and stepwise refinement should be used.

***Analysis***

Assumptions we will make are:

only integers are entered at the keyboard

only two numbers are entered

the smaller will be subtracted from the larger

the numbers are not equal

***Inputs***

two numbers

***Outputs***

a statement of the difference between the numbers

**2. Top Level Design**

1. Accept numbers
2. Difference is bigger - smaller
3. Output difference

Refinement of subproblem 2

2.1 if first is greater than second then
2.2 difference is first - second
2.3 else
2.4 difference is second - first
2.5 endif

3. First let us look at the suitability of the language we will use to code the program.

We have decided to let the numbers be integers. Can we represent the data type - integer - in Pascal? Yes

4. We want the program to be readable and understood by people new to computers - hence we will choose meaningful names for the variables.

firstnumber# = the first number

secondnumber# = secondnumber

5. How will we test that the program is correct - producing expected output?

We will test with three sets of correct data and consider what results we expect.

(i) 34 87 (ii) 1 89 (iii) 199 198

**Questions**

1. The program above is numbered 1 to 5. What stage of the software cycle does each numbered section represent?
2. In each case will the difference be first - second or second - first?
3. Design a friendly introduction screen.
4. Suggest some comment which will be helpful to the user and identify where it will go.
5. Name three modules in the program.
6. What calculations take place in the module labelled diff = first - second?
7. What decision has to be made in one of the modules?
8. What data structure is used in the second module?
9. Describe an input which could cause an error report from the system to the user.

**Answers to questions in Outcome 1**

Q1.(a)

1. Analysis, design, implementation, testing, documentation, evaluation and maintenance.
2. Clear and precise description of the problem.
3. Program specification is a clear statement of the problem with inputs, outputs and any assumptions stated.
4. Errors should be identified early in the development cycle.
5. It will cost more.
6. It is the third major revision of the program.
7. There is constant review of each stage of the development program.

Q1.(b)

1. Methodical design techniques like the use of pseudocode.
2. It is the breakdown of problem into smaller problems.
3. It makes smaller problems easier to design a solution and test.
4. Stepwise refinement is the successive breakdown of smaller problems.
5. It saves time.

Q1.(c)

1. Customer could have pressed the button beside the £200 instead of the one beside the £50. When the customer had pressed the button a message asking him to confirm that he wanted £200 or the option to return to the 'choice' menu could have been given.
2. A 'sound' alert could have been given if the money was not uplifted within a pre-set time after removal of the card.
3. 'Reliable' means correct results are given for all inputs.

**Additional questions**

1. (i) problem specification (ii) Top level design with stepwise refinement (iii) choosing a High Level Language which will support the data structures (iv) maintainability (v) choosing test data.
2. (i) second - first (ii) second - first (iii) first - second.
3. A clear message with choices.
4. Module 2 - a description of what the module will do.
5. Three modules 1. Accept numbers 2. Difference is bigger - smaller 3.Output difference.
6. The second number is taken from the first.
7. Which number is bigger.
8. Selection structure.
9. Entry of a letter.

## SOFTWARE DEVELOPMENT PROCESS

## OUTCOME 2

### Modularity

You have met the idea of breaking down a large problem into smaller problems. These smaller problems are called modules or procedures. This top down method will give you a range of easier problems to solve.

Good programming style usually means the solution has been designed from the top down. The top level program will be short. From within the top level program each procedure or module will be called. The way the procedure is called will depend on the language you are using.

Once a problem has been broken down it is easier to see what each part will do. It means that designs for the solution of a module or procedure can be produced more easily and tested.

Eventually all the modules will have to be put back together.

Finally the whole program is tested.

Ideally a module should only tackle one task. Each module should be tested separately. The modules have to share data with each other so they should be tested to see how they will work together. A small module is easier to test and debug.

The module to test for acceptable input to a program will be used many times. Once this module has been designed and tested it makes sense to store it for use in other programs. The advantage of having a library of modules can be seen. Another advantage of a module is that it could be called anywhere in a program - any number of times. All these features of a module save time in the program development.

The combination of all these module designs produces a design for the solution of the original problem.

### Control

#### *Selection*

A computer will run through a program from the beginning to the end following one instruction after another.

Suppose we want to use it to test a password (for a Bank Automatic Telling Machine perhaps) so that customers can be identified as legitimate users. Here we want the computer to make a decision. If the correct password is entered we want to give a positive message to the users. Otherwise we want to ask them to reenter the password.

There are only two possibilities.

So how do we make the computer choose?
We can use a control structure : if-then-else.

What comes between the 'if' and 'then' is a comparison statement. Here is a list of comparators used.

| | |
|---|---|
| < | is less than |
| > | is greater than |
| = | equals |
| <= | is less than or equal to |
| >= | is greater than or equal to |
| <> | is not equal to |

Going back to the question of the password you should see that the use of a selection structure would allow the computer to take two alternative paths.

```
if          password = 'user'     then
            present user option
            screen
else
            ask user to reenter
endif
```

A computer will always carry out instructions in the order given. It never gets bored and it will process a program with any number of instructions.

For instance - The problem: Get a name, address and town and print them.

This problem could be broken down by Top Down Design into four subproblems.

1. get name
2. get address
3. get town
4. print name, address, town

What if we want to print 10 sets of personal details like those above?

When you think about this problem you see that the four steps listed above will be carried out ten times. Do we write forty lines? That would be a waste of our time and the computer's. But the program can only carry out the instructions in the order given.

What would help us with this problem would be some means of getting the computer to repeat the four steps ten times.

This is where structured programming helps. One control structure we can use is the for - next loop.

***Repetition***

The for - next loop looks something like this.

```
for     variable := start to finish do
        statement
next    variable
```

The variable is used to store the number of the count. Its type should be identified. Since counts use integers {1,2,3,……..} the variable should be identified as an integer. (Frequently the # symbol is used to denote an integer). Integers need less memory to be stored than a real number ( a number like 3.23). It is more efficient to store the counter in a loop as an integer. Recall that efficiency was discussed in Outcome 1. There are two things to think about with regard to efficiency - the amount of memory used and the demands on processor time.

In the illustration of the loop

| | |
|---|---|
| statement | means one or more instructions to the computer |
| start and finish | these are called the loop counters. They are fixed integer numbers |

The loop will execute with variable holding the value 'start' first time round the loop.

We will now go back to our problem of getting the computer to write 10 address labels.

Our loop structure will be:

```
for     counter# := 1 to 10 do
        get name
        get address
        get town
        print name, address, town
next    counter#
```

Write the top down design which will write the numbers 1 to 10 in reverse order. (Hint: You can use downto to get the computer to count down.)

**Problem**

Numbers will be entered at the keyboard. A running total will be kept. This will be repeated until the total is more than 100.

In this case we can see a repeated sequence of steps. Can we use a for - next loop?

You should by now have worked out that we cannot use a *for - next* loop for this problem because we do not know the number of times it will be executed.

First we will choose a variable which will keep a count of the total. We will call this total#. What assumption does this mean we are making about the numbers that will be entered?

total# must be set to zero. It must be assigned a starting value.

```
total# := 0
    while     total# is not greater than 100
              get a number
              total# := total# + number
 endwhile
```

A new loop structure is used in this top down design. This is a conditional loop - the condition which will be tested is:

total# is not greater than 100

Look at the statement

total# := total# + number

The integer variable has already been assigned the value 0 before execution of the loop begins. Do you remember why it has been typed as an integer? So what new value will total# take if the number 19 is entered?

We could draw up a trace table to check the number of times the loop is executed and the values that total# takes.

The numbers which will be input are 7, 14, 39, 62

| number of executions of loop | value of total# |
|---|---|
| 0 | 0 |
| 1 | 7 |
| 2 | 21 |
| 3 | 60 |
| 4 | 122 |

After the fourth execution of the loop total# is now greater than 100. So when the condition - total# is not greater than 100 - is tested at the start of the loop the condition is not true so the while loop will not be executed.

Draw up trace tables for the above problem using these sets of data.

(i) 41, 59 (ii) 1, 2, 3, 5, 7, 100 (iii) 103

***Question***

The computer has to take in a number and add it to the previous. This will continue while the total is not greater than 100. Write a top down design for this.

Here is another problem which we will solve with another kind of conditional loop.

A positive number has to be entered. If the number is less than 0 then a message will be given and the user will get the opportunity to reenter a number. When the positive number is entered it has to be printed.

```
repeat
        get number
        if number is less than 0      then
        write 'try again'
        else
        write the number
until   number is greater than 0
```

The condition that is tested is ‘number is greater than 0’.

In this program segment note that the condition is tested at the end of the loop?

Below is a trace table showing for the inputs -4,-1,0.

| number of executions of loop | ‘number is greater than 0’ | message | number entered |
|---|---|---|---|
| 1 | false | try again | -4 |
| 2 | false | try again | -1 |
| 3 | true | 0 | 0 |

Draw up similar trace tables for this program and use these sets of test data.

(i) 3 (ii) -1,12 (iii) -5, -99, 67

If you are going to use a simple conditional loop try to use a while loop rather than a repeat loop. A repeat loop will always be executed at least once because the condition is not tested until the end of the loop. A while loop may not be executed at all. A while loop will use less processing power.

***Question***

1. Write a small segment of program which includes a conditional loop. Try to structure it with both a repeat loop and a while loop. Make up three sets of test data and show by means of a trace table the correctness of the program.

2. A segment of program is listed below. It represents a search for the name ‘John’ in a list.

```
FOR     pupil := 1 to 10     DO
        INPUT name$
IF      name$ = ‘John’       THEN
        found := TRUE
END IF
NEXT    pupil
```

This program is inefficient. Can you explain why? What kind of variable should ‘pupil’ be?

This section on control structures has included the use of sequence, selection and repetition in programming.

**Data Storage**

In developing a software solution to a problem we need to look at the data objects which we will use. We must think too of the operations which we should like to perform on these data objects. The type of data object we want should be given some thought.

If our problem was concerned with numbers then somehow we need to be able to represent these numbers in the memory of the computer. Different types of numbers will use different amounts of memory. Real numbers (like 3.14, 7.009, -36.9) need more memory than integers (positive or negative whole numbers like 45, -987, 2).

Some of the operations we may want to do on numbers include addition (+), subtraction (-), multiplication (*) and division (/). Notice the symbols which are used for the operations of multiplication and division.

These operations can be carried out on all types of numbers, for example:

real numbers 36.6 +23.1 = 59.7; 2.3-1 = 1.3; 2.4*0.1 = 0.24; 45.5/5 = 9.1

integers 2+89 = 91; 31-23 = 8; 45*7 = 315; 63/7 = 9

Beware - you cannot divide a number by zero!!!!! One way to avoid this possibility arising in calculations in a program care is to check all inputs to a program.

| real number | integer |
|---|---|
| number1 := 34.1 | change# := 45 |
| highest := 459.9 | highest# := 13 |
| lowest := 12.2 | gift# := 13 |

The # symbol shows an integer.

We can also compare numbers.

```
if   numberstored >25        then
     first course of action
else
     second course of action
end if
```

Certain functions like SUM (adds numbers), MAX (finds maximum number in a list of numbers, AVE (finds average of a list of numbers) can be performed on numbers.

Different types of data are stored in the computer. So far we have looked at the storage and processing of numeric data.

Another type of data object is called alphanumeric or string. Think of a program which was going to keep personal information about employees in a company. A number of string variables would be needed -for example - name, address, town.

***string variables***

name$:= john smith
address$ := 3 ayr road
town$:= Mart

String variables can be recognised by the $ symbol.

So what kind of operations can we carry out on strings. We obviously cannot carry out operations like addition, multiplication, subtraction, division, comparison or apply numeric functions.

We could join strings together.

For instance

| | |
|---|---|
| firstpart$=‘happy’ | secondpart$=‘birthday’ |
| firstpart$ + secondpart$ | = ‘happy birthday’ |

We could calculate the length of a string by applying the LEN function as shown.

| | |
|---|---|
| name$ = ozymandias | LEN(name$) = 10 |

You will get practice using strings in Outcome 3.

## SOFTWARE DEVELOPMENT PROCESS

### OUTCOME 3

Software development is the name given to the process of:

- analysing a problem
- designing a solution which will involve the use of a computer
- implementing the solution
- testing the solution
- evaluating the solution.

The following notes and tasks will help you to learn software development.

**TASK 1: YOUR FIRST PROGRAM**

Copy the following program exactly and then save, compile and run it.

```
Program CourseLength ( input,output);
{This program calculates how many classes it}
{Should take to complete this course topic}

const
   minutes = 2400 ;          {how many minutes are in a 40hr course}
var
   name : string;            {name to store the users' name}
class,course : integer;      {class to store how long}
                             {a class is in minutes}
                             {course to store how many}
                             {classes the course will last}

procedure Get_User_Info;
{This procedure prompts the user for their name and how many minutes there are}
{in a class. And then uses the variables name and lesson to store this data}

begin
  writeln('Please enter your name and then press the return key');
  readln(name);
  writeln('Please enter how many minutes per class ');
  writeln(' and then press   the return key');
  readln(class);
end;

procedure Calculate_Course_Length;
{This procedure calculates how many classes it will take to complete the course}

begin
  course := minutes / class;
end;

procedure Display_Course_Length;
{This procedure displays a message to the user on screen}
{which tells them how many classes this part of the course will take}
begin
  writeln('Hello ',name);
  writeln('This course will take ',course:3,' classes to complete');
end;

begin                        {The beginning of the program}
  Get_user_Info;
  Calculate_Course_Length;
  Display_Course_Length;
end.              {The end of the program}
```

Once you have got the program to compile and run successfully, make a print out of it and look carefully at the code.

**Specification**

The problem given to the programmer was:
Write a program that will calculate how many classes it would take to complete a 40hr course. The program will prompt the user for their name and the number of minutes in a class, calculate how many classes will be required and then display the result on screen.

This is called the Program Specification.

**Analysis**

In the analysis stage the programmers tries to write a clear and concise explanation of what the program will do and identify any Inputs, Outputs and Processes that are required.

The program will ask the user for their name and the length of a class in minutes.
It will then calculate how many lessons are needed to complete the course by dividing 2400 (40hrs = 2400 minutes) by the length of class entered. It will then display a personal message to the user that will tell them how many lessons are required.

Input: name, class length
Process calculate course length
Output name, number of classes.

**Design**

To achieve this the programmer broke the problem down by identifying the main steps that had to be taken and they were:
1. Getting the users' information
2. Calculating course length
3. Displaying course length.

This is called the Top Level Design.

Each one of these steps in the Top Level Design can then be broken down into smaller steps. And if necessary each of these can be broken down again until the problem is turned into a series of small instructions that can be translated into commands in a programming language.

This process is called stepwise refinement and for this problem it would look like this:

Getting the users' information
    Prompt for the users' name
    Accept the users' name
    Prompt for the length of a class
    Accept the length of a class
Calculating course length
    Course = 2400/ class length
Displaying course length
    Display Hello user
    Display This course will take you Course classes to complete

**Implementation**

The programmer is now ready to turn his design into a computer program:

The Top Level Design shows what procedures should be in the main program.

The Stepwise Refinements show what the procedures should do.

And the Inputs and Outputs are used to identify any variables that the program will need.

Look at the example that you have been given:

The lines that are enclosed by brackets {} are comments which are used to tell other programmers (and teachers/lecturers who mark programs) what the program does. All high level programming languages have some method of providing this internal documentation but sometimes other symbols are used to indicate where they are e.g. //.

The first section of the program after the heading is the declarations part. Where the programmer gives names to any variables and constants that will be used to store values when the program is running.

In this example the name that the user keys in, the number of minutes in a class, and how many classes it will take to complete the course will all need to be stored.
In other words the inputs and outputs identified in the analysis.

At this stage the type of the data that has to be stored will also have to be identified.
In this case:

name of type string (words or collections of characters )
class course of type integer(positive and negative whole numbers)

Other types available include real (all numbers including decimals)
char (any character on the keyboard)
and boolean (things that are either true or false)

In this example the programmer also decided to declare the number of minutes in 40hrs as a constant of 2400.

```
const  minutes = 2400
```

If this is done the program cannot accidentally change the value at run time.

The next section of this example is where all of the main steps of the Top Level Design are converted into series of programming commands called procedure calls.

Procedures are a way of grouping together programming commands which together do one task. You may wish to think of a procedure call as creating a new command which causes all the instructions inside the procedure to be carried out.

In this example three procedures are called:

Get_User_Info
Calculate_Course_Length
Display_Course_Length

Each one of these procedure calls corresponds to one of the steps in the Top Level Design.

The main program in Pascal is always the last part of the code. Hence in the example above the top level design or main program is listed last.

**We will use this technique of analysis, design and implementation for all of the examples in this course.**

**TASK 2**

**Specification**

Write a program that will calculate the monthly and weekly wages for a job when given the annual salary. The program will prompt the user for the job title and the annual salary, calculate the monthly and weekly equivalents and then display the result on screen.

**Analysis**

The program will ask the user for the job title and the annual salary. It will then calculate the monthly and weekly pays by dividing the salary entered by 12 for the monthly pay and 52 for the weekly wage. It will then display a message to the user that will tell them the weekly and monthly equivalents for that job.

Input: job title salary
Process calculate monthly pay and weekly pay
Output job title, monthly pay and weekly pay

**Top Level Design**

1. Get the job title and salary
2. Calculate monthly and weekly pays
3. Display monthly and weekly pays

**Stepwise Refinements**

1. Get the job title and salary
   1.1 Prompt for the job title
   1.2 Accept the job title
   1.3 Prompt for the salary
   1.4 Accept the salary

2. Calculate monthly and weekly pays
   2.1 Monthly = salary/12
   2.2 Weekly = salary/52

3. Display monthly and weekly pays
   3.1 Display  A Job Title will earn Monthly per month
   3.2 Display  A Job Title will earn Weekly per week

**Implementation**

It should now be relatively easy to implement this design in a programming language, and you could if you wanted to do things the hard way.  Look at the design for the previous example it has almost exactly the same structure as the design for this problem. This means that the code for the implementation should look similar as well.

Load the file that you created for Task 1 and use the editing facilities available to you to create the program on the next page.

```
Program SalaryToPay(input,output);
{This program accepts a job title and an annual salary and displays}
{The monthly and weekly pay equivalents for that job}
const
  months = 12;              {How many months are in a year}
  weeks = 52;               {How many weeks are in a year}


var
  salary : integer;         {To store the salary}
  weekly,monthly : real;    {To store the monthly and weekly pays}
                            {Type real because the answer may}
                            {not always be a whole number}
  job : string;             {To store the job title}


procedure Get_Details;
{This procedure prompts the user for a job title and an annual salary}
{It then uses the variables job and salary to store them}

begin
  writeln('Please enter the job title and then press the RETURN key');
  readln(job);
  writeln('Please enter the annual salary and then press the RETURN key');
  readln(salary);
end;

procedure Calculate_Pays;
{This procedure calculates the monthly and weekly wages for the given salary}
{And stores the results using the variables monthly  and weekly}

begin
 monthly := salary/months;
 weekly := salary/weeks;
end;

procedure Display_Pays;
{This procedure displays the results of the calculations on screen}
begin
  writeln('A ',job,' will earn £',monthly:4:2,' per month before deductions');
  writeln('A ',job,' will earn £',weekly:4:2,' per week before deductions');
end;

begin             {The main program}
  Get_Details;
  Calculate_Pays;
  Display_Pays;
end.              {The main program}
```

**Testing**

Now that you have designed and implemented your program how well does it work?

One of the most important elements of the software development cycle is testing.

Does the program perform properly for all possible inputs?

What happens if the user accidentally enters a negative value for the salary?

What about a volunteer worker for a charity who doesn't get paid and would have a salary of zero?

What happens if the user accidentally entered a character instead of a number?

To ensure that a program does what it is meant to, it must be tested.

The results of these tests should be used to either:

- highlight any problems that occur when the program is run which would indicate that there were flaws in either the analysis, the design or the implementation that need to be corrected.
- or prove that the program performs perfectly in accordance to the specification.

To do this the programmer must compile a set of test data that takes into account all possible inputs.

Run the program again using the following input and keep a note of the results.

| INPUT | | OUTPUT | | |
|---|---|---|---|---|
| **JOB TITLE** | **SALARY** | **JOB** | **MONTHLY** | **WEEKLY** |
| Teacher | 21145 | | | |
| Oxfam shop | 0 | | | |
| Lottery winner | 450000 | | | |
| 30000 | Accountant | | | |
| Gambler | -10000 | | | |
| Paper boy | 48 | | | |
| Programmer | ? | | | |

**Evaluation**

The results of these tests would seem to indicate that more work has to be done on the program but what?

Do you need to make the user prompts more specific?

Could the program check that a number has been input for salary?

Is a number acceptable as a job title?

Later in this course you will learn some of the techniques used to validate or check the data that users enter. But for now it is still a good idea to design test data sets for the programs that you will write.

**TASK 3**

Design, write and test a program that will meet the following specification.
Use the example programs to help you complete this task.

**Specification**

Write a program that will convert gallons into litres. The program will prompt the user for a volume in gallons, calculate how many litres this is and display the result on screen.

N.B. 1gallon = 4.54 litres

**TASK 4**

Design, write and test a program that will meet the following specification.
Use the example programs to help you complete this task.

**Specification**

Write a program that will calculate the area and circumference of a circle.
The program will prompt the user for the radius of a circle, calculate the area and circumference, and then display the results on screen.

N.B. Use 3.14 as a value for $\pi$

Area = $\pi r^2$

Circumference = $2\pi r$

**Repetition**

So far the programs that we have written have been quite simple. Data is input, processed and the results have been output. This is fine if all you want to do is use the computer as a very expensive calculator. One of the great things about using computer is that they are supposed to be able to do the same thing over and over again without getting bored. So how do you make them do that? Look at the following program specification and design.

**Specification**

Write a program that will display multiplication tables on screen. The program will ask the user for the table they want to see, and then calculate and display the multiplications from 1 to 12 on screen.

**Analysis**

The program will ask the user for table they want to see. It will then multiply each of the numbers from 1 to 12 by that number and display the results on screen.

Input: table
Process: multiply from 1 to 12 by table
Output: Results from 1 to 12

**Top Level Design**

1. Get the table
2. Calculate the multiplications
3. Display the table

**Stepwise Refinements**

1. Get the table
   1.1 Prompt for the table
   1.2 Accept the table
2. Calculate the multiplications
   2.1 Multiply table by every number from 1 to 12
3. Display the table
   3.1 Display the result of every multiplication from 1 to 12

**Implementation**

This would be a huge program if we used the same technique as we have for the first two examples. We would need 12 lines of calculations and another 12 lines to display the table. Fortunately there is a better way of doing this. Look at the example program on the next page which uses a technique called looping to calculate and display the table.

```
Program Tables(input,output);
{This program displays multiplication tables on screen}

var
  table,answer : integer;    {table to store the table that the user wants to see}
                             {answer to store the answer to each multiplication}


procedure Get_Table;
{This procedure prompts the user for the table they want to see}
{And then uses the variable table to store this data}
begin
 writeln('Please enter the table you want to see ');
 read(table);
end;


procedure Display_Table;
{This procedure displays a multiplication table from 1 to 12 on screen}
{By using a for loop to count from 1 to 12}
{And multiplying the counter by the table value each time}

var counter : integer;            {To store the count so far}
begin
 for counter := 1 to 12 do
 begin
   answer := counter*table;
   writeln(counter:3,' times ',table:3,' is ',answer:4);
 end;
end;

begin            {The main program}
Get_Table;
Display_Table;
end.             {The main program}
```

Copy this program exactly and then save and run it. This program does the job but does it match the design? Now that you can use loops the Top Level Design and Stepwise Refinement should look like this.

**Top Level Design**

Get the table
Display the table

**Stepwise Refinements**

1. Get the table
    1.1 Prompt for the table
    1.2 Accept the table
2. Display the table
    2.1 Loop 12 times
        2.2 Calculate the multiplication
        2.3 Display the result
    2.4 End Loop

The *for loop* is a way of making a program do the same thing over and over again for a given number of times and you can even make it work backwards.

Edit the previous example by changing the line:
for counter := 1 to 12 do
to
for counter := 12 downto 1 do

Save and then test run the program to see what happens.

**TASK 5**

**Specification**

Write a program that will display a table of fuel prices on screen. The program will ask the user for the cost of a litre of fuel. This information will be used to calculate the cost of that fuel in mutiples up to 10 litres and present the results as a table on screen.

**Analysis**

The program will ask the price of a litre of fuel. It will then multiply the numbers from 1 to 10 by that number and display the results on screen in the form of a table.

Input: price
Process: multiply from 1 to 10 by price
Output: Results from 1 to 10

**Top Level Design**

1. Get the price
2. Display the table

**Stepwise Refinements**

1. Get the price
    1.1 Prompt for the price
    1.2 Accept the price
2. Display the table
    2.1 Loop 10 times
        2.2 Calculate the cost
        2.3 Display the result
    2.4 End Loop

**Implementation**

Once again you should have a program of a very similar structure that you can edit to implement the design.

Load the previous program that displayed the multiplication tables and edit it so that it looks like this.

```
Program FuelPrices(input,output);
{this program displays a table of fuel prices on screen after being given}
{The price per litre by the user}

var price,cost : integer;


procedure Get_Price;
{This procedure prompts the user for the price of a litre of fuel}
{And then uses the variable price to store this data}

begin
 writeln('Please enter the price of a litre of petrol ');
 read(price);
end;


procedure Display_Table;
{This procedure calculates the cost for multiple litres of fuel}
{and displays this information in the form of a table}

var counter : integer;              {to store the count so far}

begin
 for counter := 1 to 10 do
 begin
    cost := counter*price;
    writeln(counter,' litres costs ',cost:4);
 end;
end;

begin
Get_Price;
Display_Table;
end.
```

**TASK 6**

Design, write and test a program that will meet the following specification.
Use the example programs to help you complete this task.

**Specification**

Write a program that will display a table of the squares of all the numbers from 1 to 12 on screen.

**TASK 7**

Design, write and test a program that will meet the following specification.
Use the example programs to help you complete this task.

**Specification**

Write a program that will display a table of the squares and cubes of all the numbers from 1 to 12 on screen.

**More For Loops**

In the previous examples the program always goes round the loop the same number of times every time it is run.
This is because the to or downto part was followed by a number.
If we replace that number with a variable and ask the user to enter a value for that variable before the loop starts then it is possible to design a more user friendly and flexible program.
Look closely at the following example that uses this technique to count up the total of a series of numbers specified by the user.

```
Program ForLoop(input,output);
{This program keeps a running total of numbers entered by the user }
{And displays the final total on screen}

var howmany,total,number : integer;          {howmany numbers they}
                                             {want to add up}
                                             {number the individual numbers}
                                             {total the final total}

procedure Initialise;
{This procedure sets the total count to zero before the counting starts}

begin
  total := 0;
end;

procedure Get_Howmany;
{This procedure prompts the user to enter how many numbers they want }
{to add up and use the variable howmany to store the value given }
```

```
begin
 writeln('Please enter how many numbers you want to add up ');
 read(howmany);
end;

procedure Count_Total;
{This procedure prompts the user to enter a number the and adds it to the }
{total so far. This process is repeated the amount of times specified by the user}

var counter : integer;

begin
 for counter := 1 to howmany do
 begin
    writeln('Please enter number ',counter);
    readln(number);
    total := total+number;
 end;
end;

procedure Display_Total;
{This procedure displays the final total after all of the counting has been done}

begin
   writeln('The total of the ',how many,' numbers you entered is ',total);
end;


begin            {The program}
   Initialise;
   Get_Howmany;
   Count_Total;
   Display_Total;
end.             {The program}
```

**TASK 8**

Design, write and test a program that will meet the following specification.
Use the example programs to help you complete this task.

**Specification**

Write a program that will prompt the user to enter a number. The program will then calculate and display the factorial of that number on screen.

4 Factorial is 10  e.g. 1+2+3+4 =10.

**TASK 9**

Design, write and test a program that will meet the following specification.
Use the example programs to help you complete this task.

**Specification**

Write a program that will prompt the user to enter the number of students in a class. The program will then prompt for the height of each student in metres before calculating and displaying the average height of the students in the class.

**Conditional loops**

For loops are fine if the programmer or the user knows how many times the program will need to go round the loop but not all problems can be solved that way.

There are situations where no one can be sure how many times the loop will have to be executed and some where it might be a good idea to ignore it altogether.

For example if a running total had to be kept of a day's transactions in a shop where it would be impossible to say exactly how many people will make purchases on any given day.

In situations like this the programmer has to use a conditional loop.
These loops will only stop running and in some cases only start if a certain conditions are met.

These conditions are usually that the result of a Boolean expression is true.
e.g. The variable that will be tested is given the value 3.

| | | |
|---|---|---|
| test := 3 | | |
| test = 3 | equals | True |
| test > 1 | greater than | True |
| test < 1 | less than | False |
| test < 4 | less than | True |
| test <> 4 | not equal to | True |
| test >1 and test < 4 | two conditions and | True |
| test =1 or test = 4 | two conditions or | False |
| test >1 and test < 3 | two conditions and | False |
| test =1 or test = 4 | two conditions or | True |

Most High Level programming languages provide for two different types of conditional loops called *while* and *repeat until* loops.

**While loops**

The first ones that we will look at are *while* loops that will only execute while certain conditions hold true.

Copy the following examples and compile and run them for your self to see how these loops work.

```
Program WhileLoop1(input,output);
{This program demonstrates a while loop that will always be executed at least once}

var test : char;              {control variable that is tested}


Procedure Init;
{This procedure gives the control variable a starting value}

begin
  test := 'N';
end;

Procedure Display_Message
{This procedure displays a message continuously until the user enters a Y}

begin
 while test <> 'Y' do                                    {test the control}
 begin                                                   {the loop}
   writeln('Are you fed up with seeing this message Y/N');
   readln(test);                                         {alter the control}
 end;                                                    {the loop}
end;

procedure Sign_Off;
{This procedure displays a message to indicate that the}
{condition needed to exit the loop has been met}

begin
   writeln('Ending the loop and program because you entered a ',test);
end;

begin         {The program}
   Init;
   Display_Message;
   Sign_Off
End.          {The program}
```

```
Program WhileLoop2(input,output);
{This program demonstrates a while loop that the user can choose not to enter}

var test : char;              {control variable that is tested}


Procedure Init;
{This procedure lets the user give the control variable a starting value}

begin
  writeln('Do you want to give the message a miss Y/N');
  readln(test);
end;

Procedure Display_Message
{This procedure displays a message continuously until the user enters a Y}

begin
 while test <> 'Y' do                                    {test the control}
 begin                                                   {the loop}
  writeln('Are you fed up with seeing this message Y/N');
  readln(test);                                          {alter the control}
 end;                                                    {the loop}
end;

procedure Sign_Off;
{This procedure displays a message to indicate that the}
{condition needed to exit the loop has been met}

begin
  writeln('Ending the loop and program because you entered a ',test);
end;

begin         {The program}
  Init;
  Display_Message;
  Sign_Off
End.          {The program}
```

**Note**

In the second example the user is given the option of missing out the instructions inside the loop altogether.

Although the examples use a variable of type char to test the condition that controls the loop the other types can be used as well.

**TASK 10**

Edit one of the *while* loop programs that you have been given and use the examples of conditions to produce programs that will:

a. Display a message until the user enters a number greater than 10.
b. Display a message if the user enters a number between 2 and 6.
c. Display a message until the user enters an x or X.
d. Display a message only if the user enters the word start and quit when they enter the word finish.

These have all been simple examples to show you how while loops work now lets try writing a program that does something useful with while loops.

**Specification**

Write a program that will play a guess the number game. The program will generate a random number between 1 and 10 and ask the user to guess what it is. It will keep on asking until the user guesses correctly and display how many guesses it took once this is achieved.

**Analysis**

The program will generate a random number between 1 and 10 using the Random Function.

It will then prompt the user to guess what the value of that random number is and then compare the two.

If it is incorrect one will be added to the total of guesses and the prompt will be repeated, otherwise the total of the guesses will be displayed on screen and the program will end.

Input: guess
Process: compare guess with number if wrong add 1 to total guesses
Output : random number, total guesses

**Top Level Design**

1. Setup the game
2. Play the game
3. Display the results

**Stepwise Refinements**

1. Setup the game
    1.1 Generate random number
    1.2 Initialise count and guess
2. Play the game
    2.1 While guess wrong
        2.2 Prompt for a guess
        2.3 Accept the guess
        2.4 Add 1 to the total of guesses
    2.5 End While
3. Display the results
    3.1 Display the random number and the number of guesses

**Implementation**

```
Program GuessLoop(input,output);
{This program demonstrates a while loop where the computer}
{decides what the stopping condition will be and doesn't}
{tell the user by playing a guessing game}

var guess, target, count : integer;     {guess to store the users guess}
{target to store what they have}
{to guess}
{count to store how many}
{guesses it took}

procedure Setup_Game;
{This procedure generates the number to guess}
{and initialises the other variables}

begin
 target := Random(10);
 guess  := 11;
 count  := 0;
end;

procedure Play_Game;
{This procedure asks the user to guess the number over and}
{over again until they get it right and keeps a count}
{of how many tries they have had}

begin
 while guess  <> target do
 begin
 writeln('Guess the number between 1 and 10');
 readln(guess);
 count := count+1;
 end;
end;
```

```
procedure Display_Results;
{This procedure displays how many times it took}
{to guess the correct answer}

begin
  writeln;
  writeln('At last you took ',count:2,' shots to find ',target:2);
end;
begin
  Setup_Game;
  Play_Game;
  Display_Results;
end.
```

**Repeat loops**

The other conditional loop that is usually available is the *repeat until* loop which keeps on repeating until a condition becomes true.

Edit the while loop examples so that they look like those below then compile and run them for yourself to see how these loops work.

```
Program RepeatLoop1(input,output);
{This program demonstrates that a repeat loop will always be executed at least once}

var test : char;              {control variable that is tested}



Procedure Display_Message
{This procedure displays a message continuously until the user enters a Y}

begin
  repeat
    writeln('Are you fed up with seeing this message Y/N');
    readln(test);                    {alter the control}
  until test = 'Y'                   {test the control }
end;

procedure Sign_Off;
{This procedure displays a message to indicate that the}
{condition needed to exit the loop has been met}

begin
  writeln('Ending the loop and program because you entered a ',test);
end;

begin         {The program}
  Display_Message;
  Sign_Off;
end.          {The program}
```

```
Program RepeatLoop2(input,output);
{This program demonstrates that the user cannot choose not to enter a repeat loop}

var test : char;              {control variable that is tested}


Procedure Init;
{This procedure lets the user give the control variable a starting value}

begin
  writeln('Do you want to give the message a miss Y/N');
  readln(test);
end;

Procedure Display_Message
{This procedure displays a message continuously until the user enters a Y}

begin
  repeat
    writeln('Are you fed up with seeing this message Y/N');
    readln(test);                    {alter the control}
  until test = 'Y'                   {test the control }
end;

procedure Sign_Off;
{This procedure displays a message to indicate that the}
{condition needed to exit the loop has been met}

begin
  writeln('Ending the loop and program because you entered a ',test);
end;

begin        {The program}
  Init;
  Display_Message;
  Sign_Off
End.         {The program}
```

**Note**

In the second example that the option of missing out the instructions inside the loop altogether does not work.

Once again, although the examples use a variable of type char to test the condition that controls the loop, the other types can be used as well.

**TASK 11**

Edit one of the repeat until loop programs that you have been given and use the examples of conditions to try and produce programs that will:

a. Display a message until the user enters a number greater than 10.
b. Display a message if the user enters a number between 2 and 6.
c. Display a message until the user enters an x or X.
d. Display a message only if the user enters the word start and quit when they enter the word finish.

**While or Repeat Until**

The differences between the loops become apparent when you try to use a *repeat until* loop to code the examples above.

It is possible to make a *while* loop act like a *repeat until* loop but if the code has to executed at least once it is easier to code the *repeat until* loop.

The design for the guess the number game using a repeat until loop will look like this:

**Top Level Design**

1. Setup the game
2. Play the game
3. Display the results

**Stepwise Refinements**

1. Setup the game
   1.1 Generate random number
   1.2 Initialise count
2. Play the game
   2.1 Repeat
      2.2 Prompt for a guess
      2.3 Accept the guess
      2.4 Add 1 to the total of guesses
   2.5 Until guess correct
3. Display the results
   3.1 Display the random number and the number of guesses

**Implementation**

```
Program GuessLoop2(input,output);
{This program demonstrates a repeat loop where the computer}
{decides what the stopping condition will be and doesn't}
{tell the user by playing a guessing game}

var guess, target, count : integer;      {guess to store the users guess}
                                         {target to store what they have}
                                         {to guess}
                                         {count to store how many}
                                         {guesses it took}


procedure Setup_Game;
{This procedure generates the number to guess}
{and initialises the count}

begin
 target := Random(10);
 count  := 0;
end;

procedure Play_Game;
{This procedure asks the user to guess the number over and}
{over again until they get it right and keeps a count}
{of how many tries they have had}

begin
  repeat
    writeln('Guess the number between 1 and 10');
    readln(guess);
    count := count+1;
  until guess = target;
end;

procedure Display_Results;
{This procedure displays how many times it took}
{to guess the correct answer}

begin
  writeln;
  writeln('At last you took ',count:2,' shots to find ',target:2);
end;
begin
  Setup_Game;
  Play_Game;
  Display_Results;
end.
```

**TASK 12**

Design, write and test a program that will meet the following specification. Use the example programs that you have been given to help you and think carefully before deciding which type of loop to use.

**Specification**

Write a program that will test the user's mental arithmetic. The program will randomly generate two numbers between 1 and 50 and then ask the user for the answer to the sum of them. If the user gets it wrong they will asked again and a tally of how many attempts it took will be kept.

**TASK 13**

Design, write and test a program that will meet the following specification. Use the example programs that you have been given to help you and think carefully before deciding which type of loop to use.

**Specification**

Write a program that will ask the user if they want to play the mental arithmetic game and if they don't it will then ask them if they want to play the guess the number game. The user can if they wish opt to play both games one after the other.

**More conditionals**

So far all of the programs that we have written have only been able to carry out sequences of instructions one after the other.

Conditional loops let the user or the programmer decide whether or not a loop will be executed and how many times it will repeat, but these are the only decisions that can be made.

Programs of any reasonable size and complexity usually have to take different courses of action depending on the data they are presented with.

To enable programs to do this most High Level programming languages provide for the use of *if* statements where values are compared and the course of action taken will depend on the result of the comparison.

To enable this comparators are provided to help compare the values with each other (see the table below).

**Comparators**

| | |
|---|---|
| = | is equal to |
| < | is less than |
| > | is greater than |
| <> | is not equal to |
| <= | is less than or equal to |
| >= | is greater than or equal to |

To see how these if statements work copy, compile and run the following program.

```
Program IfStatement1(input,output);
{This program demonstrates the use of comparators in an if statement}
var test : integer;                    {variable to test}

procedure Get_Number
{This procedure gets a number to test with an if statement}
begin
  writeln('Please enter a number to test');
  read(test);                          {Get anumber to test}
end;

procedure Test_Number
{This procedure tests a number using an if statement}
begin
   If test > 10 then                                    {See if it's bigger than 10}
     writeln('That number was a bigger than ten ');     {Tell the user if it is}
                                                        {Do nothing if it isn't}
end;
```

```
begin               {The Program }
  Get_Number;
  Test_ Number
end.
```

Try editing this program to use the other comparators and alter the message that is displayed to give an appropriate comment.

**Else**

The previous programs would be OK if we did not want to do anything if the comparison failed (while loops can do that) but what happens if you want to do something else.

The second, and optional, part of if statements provides the option to do something else if the comparison fails.

Copy, compile and run the following program to see how else can be used to make programs more flexible.

```
Program IfStatement2(input,output);
{This program demonstrates the use of an if and else statement}
var test : integer;                    {variable to test}

procedure Get_Number
{This procedure gets a number to test with an if else statement}
begin
  writeln('Please enter a number to test');
  read(test);                          {Get anumber to test}
end;

procedure Test_Number
{This procedure tests a number using an if else statement}
begin
  if test > 10 then                                  {See if it's bigger than 10}
    writeln('That number was a bigger than ten ');   {Tell the user if it is}
  else
    writeln('That number was less than ten');        {Or is not}
end;

begin               {The Program }
  Get_Number;
  Test_ Number
end.
```

Things are starting to look a little bit more friendly now that we can choose which message to display depending on what the user enters at the prompt.

But the best is yet to come because the else part can be an instruction to do another if.

So in effect the instruction becomes if or else if a bit like playing twenty questions.

Copy, compile and run the following program to see how if, else, can be used.

```
Program IfStatement3(input,output);
{This program demonstrates the use of an if and else if  statement}
var test : integer;                    {variable to test}

procedure Get_Number
{This procedure gets a number to test with an if else if statement}
begin
  writeln('Please enter a number to test');
  read(test);                          {Get anumber to test}
end;

procedure Test_Number
{This procedure tests a number using an if else if statement}
begin
   if test > 10 then                                    {See if it's bigger than 10}
     writeln('That number was a bigger than ten ');     {Tell the user if it is}
   else
      if test = 10 then
        writeln('That number was exactly 10')           {Or it's exactly 10}
      else
        writeln('That number was less than ten');       {Or is not}
end;

begin                {The Program }
   Get_Number;
   Test_ Number
end.                 {The Program }
```

**PRACTICAL EXERCISE**

**Range of letters data validation**

Design and write a program which will accept an A or B as valid letters and display an error message if the input is not an ‘A’ or a ‘B’.

**Look at the ‘or’, ‘and’ examples in the while loops section. Can you write the above to accept upper or lower case as valid without adding any more ifs.**

**Chop and play with strings**

String variables are slightly different in that different operations are performed on them. You cannot multiply one by the other, divide one by another or subtract one from another. You can join them together. There are other operations you can do. You can measure them. Follow this example.

```
Program StringLength(input,output);
{This program will take in a string and calculate its length which will be output}

var
   words : string;    {to store the string}
   length : integer;  {to store the length}

procedure Get_String;
{This procedure asks for the string}
begin
writeln('Please enter the word or words to measure');
writeln('and then press the RETURN key');
readln(words);
end;

procedure Measure_String;
{This is the standard procedure len to calculate the length of the string}
begin
length := len(words);
end;

procedure Display_Length;
{This procedure displays the length}
begin
writeln('That String was ',length:3,' characters long');
end;

begin
  Get_String;
  Measure_String;
  Display_Length;
end.
```

This example uses both len and substring which chops the string

```
Program Palindrome(input,output);
{This program prompts the user to enter a string and then}
{reverses it to check if it is a palindrome}

var
   words, backwards, part : string;
   length : integer;

Procedure Get_String;
{This procedure prompts the user to enter a string to check}

begin
  writeln('Please enter the word or words to check');
  writeln('and then press the RETURN key');
  readln(words);
end;

Procedure Reverse_String;
{This procedure reverses the string that was entered}
{So that it can be checked to see if it is a palindrome}

var
   counter : integer;

begin
  length := len(words);
  backwards := ' ';
  for counter := length downto 1 do
  begin
    part := substring(words,counter,1);
    backwards := backwards + part;
  end;
  backwards := substring(backwards,2,length) ;
end;

Procedure Display_Reverse;
{This procedure displays the string entered and its reverse}

begin
writeln;
writeln(words);
writeln;
writeln('backwards is ');
writeln;
writeln(backwards);
writeln;
end;
```

```
Procedure Check_Palindrome;
{This procedure checks to see if the string is a palindrome}
{and displays an appropriate message}

begin
  if words = backwards then
    writeln('Therefore it is a Palindrome')
  else
    writeln('Therefore it is not a Palindrome');
end;


begin                {main program}
   Get_String;
   Reverse_String;
   Display_Reverse;
   Check_Palindrome;
end.                 {main program}
```

Examples of palindromes are 'mum', 'dad', 'alla', 'madam'. You can make some more yourself and try them.

This program will chop the initial letters of strings and add these together to produce the acronym.

Examples which you can test are WIMP (Windows, Icons, Menu, Pointer) RAM (Random Access Memory). Try a few more yourself.

```
Program Acronym(input,output);
var
    words, acronym, part : string;
    length : integer;

procedure Get_String;

begin
  writeln('Please enter the words to make an acronym');
  writeln('and then press the RETURN key');
  readln(words);
end;

procedure Make_Acronym;

var
   counter : integer;

begin
  length := len(words);
  acronym := substring(words,1,1);
  for counter := 2 to length do
  begin
    part := substring(words,counter,1);
    if part = ' ' then
    begin
      part := substring(words,counter+1,1);
      acronym := acronym + part;
    end;
  end;
end;

procedure Display_Acronym;

begin
writeln('The acronym for that phrase is ',acronym);
end;

begin
   Get_String;
   Make_Acronym;
   Display_Acronym;
end.
```

**Exercise**

Concatenate forename and surname entered separately.
Separate forename and surname entered as one String.

Modify both to display initials as one string.

Now that you have learnt how to use most of the features of a High Level Programming Language let us look at how they may be used to solve a slightly larger problem.

**Specification**

Write a program that will display a menu of options similar to this on the screen.

S Convert Salary
P Check Palindrome
G Play Guess the number
X Exit

When a code is entered it will be checked to see if it is valid and if not an error message will be issued and the user will be given the opportunity to try again.

**Analysis**

As long as the user has not entered an X the program will display a menu of options on screen. Depending on the input the program will either: prompt for and convert a salary into monthly and weekly pays; check a string to see if it is a palindrome; play the guess number game; or quit after a farewell message.

If the option is not valid an error message will be issued and the menu will be displayed again.

**Inputs**

| | | | |
|---|---|---|---|
| menu choice | S | P | G |
| | job title salary | string test | guesses |

**Processes**

Check option choice
calculate monthly and weekly pays
reverse words and compare with the original
compare guess with random number and add 1 to counter if failed
display farewell message and stop running

**Outputs**

monthly and weekly pays
words and their reverse with appropriate message
random number and the number of guesses
farewell message

**Top Level Design**

1. while choice not X
2. display menu
3. accept choice
4. process choice
5. end while

**Stepwise Refinement**

| | | | |
|---|---|---|---|
| 1 while | choice not X | | |
| | 2 display menu | | |
| | | 2.1 display options | |
| | | 2.2 prompt for choice | |
| | accept choice | | |
| | process choice | 4.1 if choice = X | quit program |
| | | 4.2 S | convert salary |
| | | 4.3 P | check pallindrome |
| | | 4.4 G | play game |
| | | 4.5 else | |
| | | | error |

**Implementation**

At first this may appear to be a very long program but we already have most of the procedures required to implement a solution to this program.

Cut and paste all of the procedures required into a new file and then look at what is left to do.

The programs that you wrote to solve these problems can now be turned into procedures in our big program. Look at the following example to see how to do this.

Now there are only four procedures left to write for the big program: Pause, ProcessChoice, DisplayMenu and StartMenu. Copy them from disc into your program and then compile and test run it.

```
Program MenuChoices(input,output);
{This program displays a menu of options on screen and depending}
{on the users' choice either converts a salary to weekly and monthly pays}

{checks a palindrome or plays a guess the number game}
{An error message is generated if an invalid option is entered}
{and the user is given the chance to try again}
```

```
{An X entered will end the program }

const  weeks = 52;
     months = 12;

var salary, guess, target, count, length : integer;
   weekly,monthly : real;
   job, words, backwards, part : string;
   test : char;

procedure Get_Details;
{This procedure prompts the user for a job title and an annual salary}
{It then uses the variables job and salary to store them}
begin
writeln('Please enter the job title and then press the RETURN key');
readln(job);
writeln('Please enter the yearly salary and then press the RETURN key');
readln(salary);
end;

procedure Calculate_Pays;
{This procedure calculates the monthly and weekly wages for the given salary}
{And stores the results using the variables monthly and weekly}

begin
monthly := salary/months;
weekly := salary/weeks;
end;

procedure Display_Pays;
{This procedure displays the results of the calculations on screen}

begin
writeln('A ',job,' will earn £',monthly:4:2,' per month before deductions');
writeln('A ',job,' will earn £',weekly:4:2,' per week before deductions');
end;

procedure SalaryToPay;
{This procedure prompts the user for a job title and annual salary}
{And displays the monthly and weekly equivalents for that job on screen}

begin
Get_Details;
Calculate_Pays;
Display_Pays;
end;
```

```
Procedure Get_String;
{This procedure prompts the user to enter a string to check}

begin
  writeln('Please enter the word or words to check');
  writeln('and then press the RETURN key');
  readln(words);
end;

Procedure Reverse_String;
{This procedure reverses the string that was entered}
{So that it can be checked to see if it is a palindrome}

var
   counter : integer;

begin
  length := len(words);
  backwards := ' ';
  for counter := length downto 1 do
  begin
    part := substring(words,counter,1);
    backwards := backwards + part;
  end;
  backwards := substring(backwards,2,length) ;
end;

Procedure Display_Reverse;
{This procedure displays the string entered and its reverse}

begin
writeln;
writeln(words);
writeln;
writeln(' backwards is ');
writeln;
writeln(backwards);
writeln;
end;

Procedure Check_Palindrome;
{This procedure checks to see if the string is a palindrome}
{and displays an appropriate message}

begin
  if words = backwards then
    writeln('Therefore it is a palindrome')
  else
```

```
writeln('Therefore it is not a palindrome');
end;

procedure Palindrome;
{This procedure prompts the user to enter a string and then}
{reverses it to check if it is a palindrome}

begin
  Get_String;
  Reverse_String;
  Display_Reverse;
  Check_Palindrome;
end;

procedure Setup_Game;
{This procedure generates the number to guess}
{and initialises the other variables}

begin
 target := Random(10);
 guess  := 11;
 count  := 0;
end;

procedure Play_Game;
{This procedure asks the user to guess the numberover and over again}
{until they get it right and keeps a count of how many tries they have had}

begin
 while guess  <> target do
 begin
 writeln('Guess the number between 1 and 10');
 readln(guess);
 count := count+1;
 end;
end;

procedure Display_Results;
{This procedure displays how many times it took to guess the correct answer}

begin
  writeln;
  writeln('At last you took ',count:2,' shots to find ',target:2);
end;

Procedure GuessNumber;
{This procedure generates a random number and asks the user to guess it}
{until they they succeed and then displays how many guesses it took them}
```

```
begin
  Setup_Game;
  Play_Game;
  Display_Results;
end;

procedure Pause;
{This procedure waits until the user presses any key}

var dummy : char;

begin
  writeln;
  write('Press any key to continue');
  read(dummy);
  page;
end;

procedure Process_Choice;

begin
  if (test = 'x') or (test = 'X') then
   begin
     writeln;
     writeln('Quiting Program Now Bye Bye');
   end
  else
      if (test = 's') or (test ='S') then
       begin
         page;
         SalaryToPay;
         pause;
       end
      else
        if (test = 'p') or (test ='P') then
          begin
            page;
            Palindrome;
            pause;
          end
        else
          if (test = 'g') or (test ='G') then
            begin
              page;
              GuessNumber;
              pause;
            end
          else
            begin
```

```
            page;
            writeln;
            writeln('You entered a ',test,' please enter a valid option');
            writeln;
            writeln;
          end;
end;

procedure Display_Menu;
begin
  writeln('S   Convert a Salary');
  writeln('P   Check a Palindrome');
  writeln('G   Play Guess the Number');
  writeln('X   Exit');
  writeln;
  writeln;
  write('Please enter your choice and press Return :- ');
  readln(test);
  process_choice;
end;

procedure Start_Menu;
{This procedure initialises the control variable}
{and displays the first menu to the user}

begin
  test := 'A';
  while (test <> 'x') or (test <> 'x')do
    begin
      Display_Menu;
    end;
end;

begin
  Start_Menu;
end.
```

## SOFTWARE DEVELOPMENT PROCESS

## OUTCOME 4

### Problem solving

The following problem will let you practise all your skills in software development. It is similar to the kind of problems that you will tackle for the unit assessment.

***Task 14***

Write a program which will display a menu of options on the screen. The choices are as shown below.

1 Multiplication Table
2 Make Acronym
3 Test Mental Arithmetic
4 Exit

When a code is entered it will check to see if it is valid. An error message will be given if the code is not valid and the user will be prompted to try again. Use what you have learned in the previous programs to help you. Your program should include a statement of specification, analysis, inputs, processes, outputs, top level design with stepwise refinement, program listing and runs, test data, documentation, review.