Software DEsign and development

National 5 Computing Science

Contents

[Development Methodologies 4](#_Toc482217581)

[Analysis 5](#_Toc482217582)

[Assumptions 5](#_Toc482217583)

[Inputs 5](#_Toc482217584)

[Processes 5](#_Toc482217585)

[Outputs 5](#_Toc482217586)

[Design 6](#_Toc482217587)

[Data types and structures 6](#_Toc482217588)

[Structure diagrams / Flowcharts 8](#_Toc482217589)

[Pseudocode 8](#_Toc482217590)

[Efficient solutions 10](#_Toc482217591)

[User interface design 11](#_Toc482217592)

[Implementation (data types and structures) 12](#_Toc482217593)

[Data types and structures 12](#_Toc482217594)

[One dimensional (1-D) arrays 12](#_Toc482217595)

[Implementation (computational constructs) 14](#_Toc482217596)

[Assignment statements 14](#_Toc482217597)

[Arithmetic operators 14](#_Toc482217598)

[Concatenation 14](#_Toc482217599)

[Inequalities 15](#_Toc482217600)

[Conditions 15](#_Toc482217601)

[Logical operators 15](#_Toc482217602)

[Selection constructs (IF … THEN … ELSE … END IF) using conditions 15](#_Toc482217603)

[Fixed loops 16](#_Toc482217604)

[Conditional loops 16](#_Toc482217605)

[Pre-defined functions and parameters 17](#_Toc482217606)

[Implementation (algorithm specification) 18](#_Toc482217607)

[Output 18](#_Toc482217608)

[Input validation 18](#_Toc482217609)

[Running total (using a loop) and finding the average 18](#_Toc482217610)

[Traversing a 1-D array: Linear search 18](#_Toc482217611)

[Traversing a 1-D array: Counting Up 19](#_Toc482217612)

[Nested loops 19](#_Toc482217613)

[Testing 20](#_Toc482217614)

[Test data 20](#_Toc482217615)

[Errors 20](#_Toc482217616)

[Evaluation 21](#_Toc482217617)

[Fitness for purpose 21](#_Toc482217618)

[Efficient use of coding constructs 21](#_Toc482217619)

[Robustness 22](#_Toc482217620)

[Readability of code 22](#_Toc482217621)

# Original version courtesy of

Alison Cowie

Portobello High School, Edinburgh

## Development Methodologies

Developing a piece of software usually has six phases. These are:

**1. Analysis**

**2. Design**

**3. Implementation**

**4. Testing**

**5. Documentation**

**6. Evaluation**

For example, Andrea is a Project Manager at AppsForLawyers. Jenny is a manager in a law firm and has asked Andrea at AppsForLawyers to develop a piece of software that will automate some legal documents for her firm.

***Analysis***: Andrea and Jenny will meet to decide exactly what Jenny wants the software to do. They will create a document called the **software specification** that lists the **functional requirements** of the software. This is often done by identifying the **inputs, processes and outputs** of the software.

***Design***: Andrea will pass the software specification to her design team who will **plan the** **data structures** to hold the data that will be processed by the software. They will also **plan the user interface** and they will **plan the modules of program code** that will be required.

**Iteration**:Jenny may be involved in checking the design and she may ask for changes at the design phase based on what she sees and this could mean going back to the analysis phase with Andrea. **Iteration** means ‘to do over again’. The software development process is described as **iterative** because **each stage may have to be revisited as a result of new information coming to light.**

***Implementation***: The design is passed to the programmers who **write program code to implement the functional requirements of the software**. The programmers may create **prototypes** for Jenny to see. If the programmers need the design clarified for them, Jenny and Andrea may have to meet with the design team again (iteration). This may even require them to go back to the original analysis (iteration).

***Testing***: The **programmers will be testing the code continually** while they are working. When they are finished, Andrea will make up a test team from AppsForLawyers who can test the software as a whole. They probably won’t know the actual code in detail – this is sometimes called **alpha testing**. Once again, this may throw up issues that require previous phases to be revisited (iteration). For example, the programmers may have to re-code some parts. Finally, the software will be passed to Jenny’s law firm for **beta testing.**

***Documentation***: This includes a **technical guide** and a **user guide**. The technical guide explains **how to install the software** and **what the system requirements are** e.g. operating system, amount of memory etc. Theuser guide will **help the user to use the software successfully.**

***Evaluation***: This is to decide whether the software is **fit for purpose**. That means: Does it meet its **functional requirements**? Is it **robust** (runs successfully without crashing)? Is it **readable** code in case people need to edit it?

## Analysis

In reality, for big software development projects, the analysis phase can take months to complete. Let’s look at a *much* simpler example:

Mrs Bird is taking 15 students on a school trip. The cost of the trip is £50 but the students can pay by installment. Mrs Bird has called a meeting to take the **first** installment and she wants a piece of software (a program) to let her enter the amount paid by each student. At the end of the meeting she would like to be able to display how much each student has left to pay. She also wants to know how many students still have the full £50 left to pay

The analysis for this program might look like this:

### Assumptions

Assumptions often arise as the analysis proceeds. For example, “I will assume …”

* “all 15 students are at the meeting”
* “some students may have forgotten to bring the money with them”
* “that the student’s name will need to be input as well as the amount they pay”

### Inputs

Each student’s name (15 students) and store

The amount paid by each student

### Processes

Validate the amount paid in the range 0 to 50 inclusive

Subtract the amount paid from 50 and store

Count up how many students still have £50 left to pay

### Outputs

Display list of names with amount still owing

Display number of students who still have the full £50 left to pay

## Design

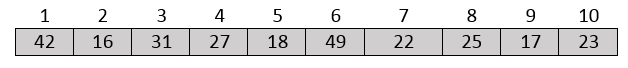
### Data types and structures

Example 1: You are asked to write a program that finds the **average** age of ten people and **whether** any of the 10 people are of average age

One of your first steps would be to *draw* and *define* the data structures (the holding places) for the data that will be used. You should include example data or test data in the diagram.

age

*age : array[1..10] of integer*



The variables are stored as follows:

**total**

270

**averageAge**

27

**anyOfAverageAge**

TRUE

*total :* ***integer***

*averageAge* ***: integer***

*anyOfAverageAge :* ***boolean***

Example 2: The data structures for Mrs Bird’s school trip (see page 5)

|  |  |
| --- | --- |
| 1 | “Bilal”  studentName |
| 2 | “Harpreet” |
| 3 | “Joshua” |
| 4 | “Kieran” |
| 5 | “Lucy” |
|  | .  .  . |
| 12 | “Paolo” |
| 13 | “Robin” |
| 14 | “Yousif” |
| 15 | “Zach” |

|  |  |
| --- | --- |
| 1 | 25.0  amountLeft |
| 2 | 35.0 |
| 3 | 20.0 |
| 4 | 16.75 |
| 5 | 50.0 |
|  | .  .  . |
| 12 | 40.0 |
| 13 | 0.0 |
| 14 | 50.0 |
| 15 | 15.0 |

stillToPayFull

2

stillToPayFull *:* ***integer***

*amountLeft :* ***array****[1..15] of real*

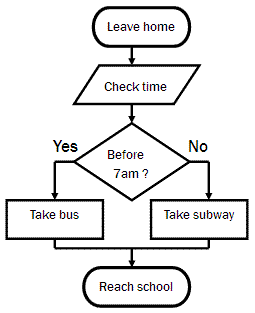
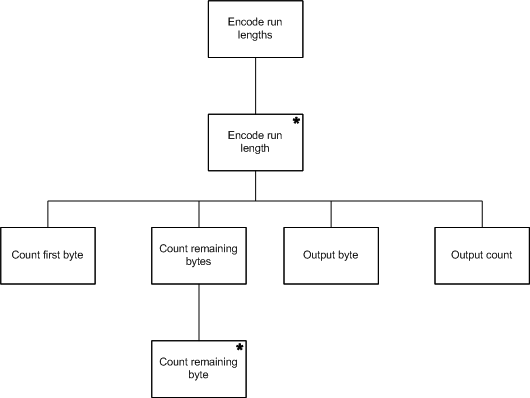
*studentName :* ***array****[1..15] of string*

**Data structures** are the groups of storage locations in memory that will be used by the software. The choice of name is very important – the names of the data structures can significantly help people read and understand the code

Once the data structures have been defined (e.g. **array**), the modules of code are planned out. This might be in the form of a diagram like a flowchart or a structure diagram or it might be in pseudocode

### Structure diagrams / Flowcharts

This uses specially shaped boxes to show loops, decisions etc. in the program. The boxes are linked together to show how the program fits together. Apart from **flow charts** and **structure diagrams**, other graphical design methods include linked charts and block diagrams



**Advantages:** the structure and program flow is very clear

**Disadvantages:** adding more detail means the diagrams can become very complicated

### Pseudocode

Pseudocode is an English-like language used to define problems. It can be laid out to reflect the structure of the actual program

**Advs: *close to English >> easy to understand***

***easy to stepwise-refine the design*** (break the steps down into smaller and smaller steps)

e.g. RECEIVE centigrade FROM KEYBOARD

SET fahrenheit TO ( 9 / 5 ) \* centigrade + 32

SEND fahrenheit TO DISPLAY

The pseudocode shown above is written in a pseudocode called Haggis. You could write this pseudocode as follows:

receive centigrade from keyboard

set fahrenheit to ( 9 / 5 ) \* centigrade + 32

send fahrenheit to display

input centigrade

fahrenheit = ( 9 / 5 ) \* centigrade + 32

output fahrenheit

or…

**Pseudocode, structure diagrams** and **flowcharts** are just some of the design tools used to ‘*think through the action of program code’* before having to actually program it in a specific high level language such as Python or Java.

Pseudocode for Mrs Bird’s program: (written in Haggis pseudocode) (see page 5 and page 7)

SET stillToPayFull TO 0

FOR studentNumber FROM 1 TO 15 DO

RECEIVE studentName[studentNumber] FROM (STRING) KEYBOARD

RECEIVE installment FROM (REAL) KEYBOARD

WHILE (installment < 0) OR (installment > 50) DO

SEND “Invalid data – out of range” TO DISPLAY

RECEIVE installment FROM (REAL) KEYBOARD

END WHILE

SET amountLeft[studentNumber] TO 50 – installment

IF amountLeft[studentNumber] = 50 THEN

SET stillToPayFull TO stillToPayFull + 1

END IF

END FOR

FOR studentNumber FROM 1 TO 15 DO

SEND studentName[studentNumber] TO DISPLAY

SEND amountLeft[studentNumber] TO DISPLAY

END FOR

SEND stillToPayFull TO DISPLAY

### Efficient solutions

Efficient solutions often involve using loops. If the same action is carried out over and over again, it would be inefficient to keep writing the same code over and over again in the program.

Suppose we wanted to add 20% to all the prices of 2000 hats on a shopping site. It wouldn’t be efficient to write code like this:

**SET hatPrice[1] TO hatPrice[1] \* 1.20**

**SET hatPrice[2] TO hatPrice[2] \* 1.20**

**SET hatPrice[3] TO hatPrice[3] \* 1.20**

**.**

**.**

**.**

**SET hatPrice[2000] TO hatPrice[2000] \* 1.20**

It would be better to write this:

**FOR hatNumber FROM 1 TO 2000 DO**

**SET hatPrice[hatNumber] TO hatPrice[hatNumber] \* 1.20**

**END FOR**

Why is the second way more **efficient**?

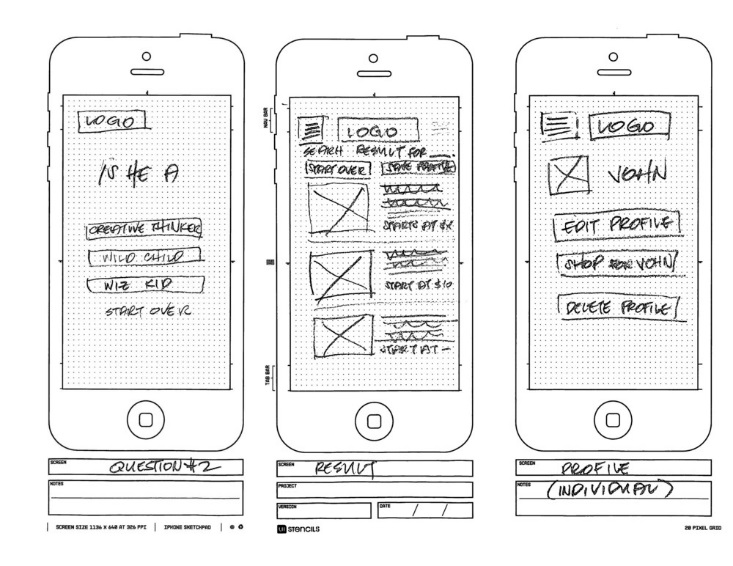
**Editing:** The second way is more efficient from an **editing** point of view (i.e. **time**). It is **easier to read** than pages of the same statement, and, if you needed to change the percentage, then *only one* line of code needs to be edited instead of 2000 lines

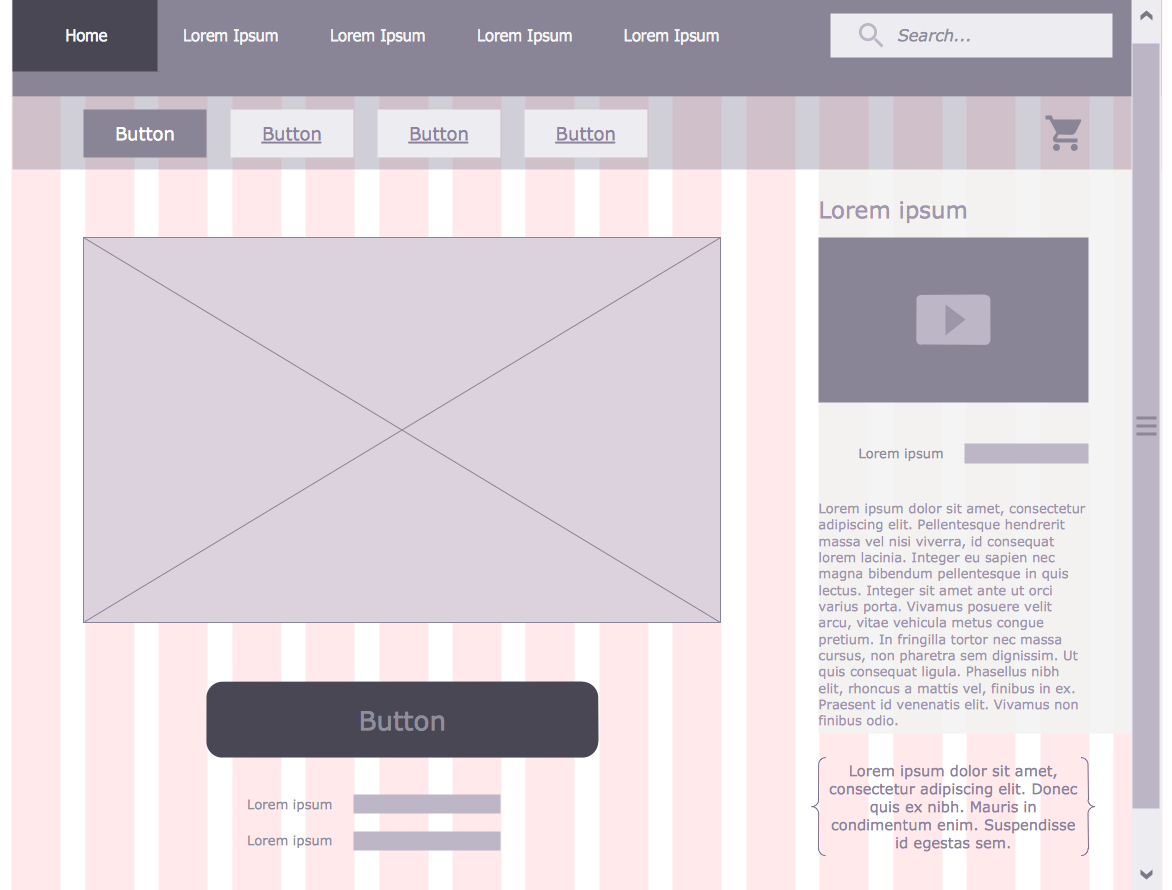
**Memory usage:** It is more efficient from a memory point of view. Each line of code is held in memory while the program is running so to have just 3 lines instead of 2000 lines of code in memory means that memory usage is more efficient

**Processor time:** A processor-time efficient program will be one where the processor is not busier than it needs to be. This is especially important in time-critical situations (like a nuclear reactor). When a program is translated into machine code, a loop can be translated into machine code that takes less processor time to run than code that’s carrying out the same task but without a loop

### User interface design

Here are some examples of wireframes of user interfaces



These are wireframe diagrams for apps

on a mobile device e.g. smart phone

This is a wireframe diagram for a webpage

## Implementation (data types and structures)

### Data types and structures

Data structures holds data for the programmer to use. They are called **variables** in programming.

Different types of structures can be created using the storage locations in memory. The simplest structures are variables that hold just one single value.

**global *count*** *//integer* (a number with no fractional part e.g. 20, -36)

**global *average***  *//real* (a number with a fractional part e.g. 65.7)

**global *keypress*** //char (a single character e.g. “R” or “\*” or “8”)

**global *name*** *//string*  (a group of characters e.g. “Allison Cowie” “EH16 5PJ”)

**global *passed*** *//boolean* (either TRUE or FALSE)

### One dimensional (1-D) arrays

An array is used when a *list* of values is required rather than one single one. It is a group of storage locations all storing **the same type of data**. The group is given a group name and an index number is used to determine which element is being referenced.

e.g. **put “0602” into studentID[4]**  *//studentID is a one dimensional array of 6 strings*

RAM

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 1 | 2 | 3 | 4 | 5  << studentID array >> | 6 |  |  |  |
| … |  |  |  |  |  | “0602” |  |  |  |  | … |

Square brackets are often used to indicate an array

**global marks** *//array[20] of integer* (a list of integers)

**put 58 into marks[1]** *//assigns the value 58 to element #1 of the array called “marks”*

**put 96 into marks[15]** *//assigns the value 96 to element #15 of the array called “marks”*

**put total + marks[10] into total**

*//adds the value of the 10th element of “marks” to the variable called “total”*

Note that some languages start their arrays at 0 rather than 1.

This means that arrays can be indexed from 0 to 9 or arrays can be indexed from 1 to 10.

For example:

**repeat with product = 0 to 9** *//loop 10 times*

**put cost[product] \* 0.50 into cost[product]**

**end repeat** *//multiplies all 10 elements of “cost” by 50%*

cost

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3.45 | 6.50 | 7.75 | 1.25 | 5.00 | 2.50 | 4.35 | 7.20 | 2.00 | 4.10 |

1-D stands for one-dimensional (For interest: 2-D would be two-dimensional)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

In the above example

cost[0] contains the value 3.45

cost[5] contains the value 2.50

The 4th value in the cost[] array is 5.00

## Implementation (computational constructs)

### Assignment statements

This is when a *value* is given to a *variable* (a variable is the name of a storage location in memory)

**put 58 into total** // total is a variable (storage location in RAM) holding the value 58

**put “Paul Sinclair McTavish” into nameUser**

*//direct assignment to the string variable called “nameUser”*

**put total / 20 into average**

*//divides the value of the variable “total” by 20 and stores the result in the variable called “average”*

**ask “please enter your name”** *//prompts the user to enter their name*

**if the result = “cancel” then exit to top** *//allows the user to exit the program*

**put it into nameUser** *//**data is input**from the keyboard*

*// the value entered by the user is stored in a string variable called “nameUser”*

### Arithmetic operators

These are: + - \* / ^

e.g. **put 1/2 \* length \* breadth into area**

e.g. **put priceOfCamera – priceOfCamera \* 0.05 into priceOfCamera**

^ is used for exponents. For example, 2^8 = 256

### Concatenation

The & symbol is used to combine strings and arrays.

For example, concatenating two strings put “hello “ & “Mary” into greeting

Here, the variable **greeting** will be assigned the value “Hello Mary”

For example, concatenating two arrays

girls = ["Cecilie", "Lone"];  
boys = ["Emil", "Tobias", "Linus"];  
cousins = ["Robin"];  
children = girls & boys & cousins;

The values of the children array will be: ["Cecilie", "Lone", "Emil", "Tobias", "Linus", "Robin"]

### Inequalities

**<>** (not equals to) **>** (greater than) **<** (less than)

**<=** (less than or equal to) **>=** (greater than or equal to)

### Conditions

**Simple condition** – just one set of brackets e.g. (cost < 15.99)

**Complex condition** – more than one set of brackets joined by logical operators

e.g. **if (cost > 15.99) AND ((age <= 8) OR (age >= 65) ) then**

e.g. **if (temperature > 100) OR NOT(state = “working”) then**

### Logical operators

These are: AND OR NOT

e.g. **repeat until (time > 0) AND (time < 20) AND NOT(time = 10)**

### Selection constructs (IF … THEN … ELSE … END IF) using conditions

**if (cost < 5) then** *//simple condition- if the cost is less than 5*

**answer “That’s quite cheap!”**

**end if**

**if (age >= 17) then** *//if the variable “age” is at least 17*

**answer “ you are old enough to drive”** *//program outputs “you are old enough…”*

**else** *//and, if the variable “age” is less than 17,*

**answer “sorry – not old enough to drive yet”** *//program outputs “sorry – not …”*

**end if**

**if (scoreOne > 60) AND (scoreTwo > 60) then** *//complex condition – if score1 and score 2 are greater than**60*

**if (age >= 10) AND (age <= 14) then** *//complex condition – if age is >=10 and <=14*

**answer “Congratulations - you are through to the Junior Final”**

**else if (age >= 15) AND (age <= 18) then** *//complex condition – if age is >=15 and <=18*

**answer “Congratulations - you are through to the Senior Final”**

**else**

**answer “You are not the right age to be in this competition!”**

**end if**

**end if**

**else**

**answer “Sorry – you didn’t score high enough to be in either Final**

**end if**

### Fixed loops

These are used when code is to be repeated an *exact number of times*

e.g. **put 0 into total**

**repeat with count = 1 to 10** *//loop for all 10 people*

**ask “What is your age?”**

**put it into ageUser**

**if ageUser >= 18 then add 1 to total**

**end repeat**

e.g. **repeat with questionNumber = 1 to 20** *//loop for all 20 questions*

**put question[questionNumber] & “?” & return after field “Questions display”**

**end repeat**

*The variable “****questionNumber****” is used as an index into the array called “****question****” e.g.* question[questionNumber]

### Conditional loops

These are used when it is *not known how many times* a block of code is to be repeated

**ask “please enter your password: “** *//prompts the user to enter a password*

**put it into password** *//stores the value entered in the variable password*

**repeat until (password = “A1rdr13”)** *//start of conditional loop*

**answer “That’s incorrect – try again”** *//error message to alert user*

**ask “please re- enter your password: “** *//prompts the user to enter password again*

**put it into password** *//stores the password entered*

**end repeat** *//end of conditional loop*

### Pre-defined functions and parameters

These are modules of code that have already been written for programmers to use. They are held in a library of pre-defined functions and save the programmer time when coding. It also means that the programmer does not need to know in detail how to write all parts of the program

Examples of some common pre-defined functions: **length**, cos, sin, div, sqrt, **round**, **random**, integer

Quite often, the programmer needs to pass some information to the module e.g. the value 35 is passed to the square root function like this: sqrt(35) In this example, 35 is called the **parameter**.

5.916079783099616

e.g. **put sqrt(35) into field “display”** ***(displays the square root of 35)***

5.92

e.g. **put round(sqrt(35), 2) into field “display”** ***(displays answer to 2 dp)***

e.g. **put random(100) into guess**

This function will assign a random number between 1 and 100 to the variable called *guess*. The parameter, 100, is in brackets

e.g. **put length(“EH15 3AS”) into numChars**

This function works out the number of characters in the parameter (text string). The parameter is the string “EH15 3AS” and the number of characters in “EH15 3AS” is 8 so the value 8 is assigned to the variable called *numChars*

## Implementation (algorithm specification)

### Output

**put “Welcome, ” & nameUser & “!” & return into field “Welcome message”**

*//outputs the word* Welcome *and the value of the string variable “nameUser” followed by an exclamation mark to the display field. The concatenation operator & is used to combine values*

**put empty into field “Student marks”**

**repeat with count = 1 to 20** *//”count” is used to process each student in turn*

**put count & “:” & tab & mark[count] & return after field “Student marks”**

**end repeat**

*//clears the display field and then outputs the number of each of the 20 students and their corresponding mark to the display field. A new line is taken between each one.*

### Input validation

**ask "How many students? " //allowing 1 to 5 students**

**if the result = "cancel" then exit to top**

**put it into numStudents**

**repeat until (numStudents >= 1) AND (numStudents <= 5)** //validating input in range 1 to 5 inclusive

**answer "Invalid number - try again"**

**ask "How many students? "**

**if the result = "cancel" then exit to top**

**put it into numStudents**

**end repeat**

### Running total (using a loop) and finding the average

**put 0 into total** *//initialises the variable “total” to 0*

**repeat with count = 1 to 20** *//loops for each of the 20 students*

**put total + marks[count] into total** //*adds each student’s mark to the total*

**end repeat** //*“count” is used as the index into the array*

**put total / 20 into average** *//calculates the average mark by dividing the total by the number of students*

### Traversing a 1-D array: Linear search

**ask “Enter your search value”**

**put it into target**

**repeat with count = 1 to 100**

**if name[count] = target then answer “Yup – that person is in the list”**

**end repeat**

### Traversing a 1-D array: Counting Up

**put 0 into total**

**repeat with count = 1 to 10**

**if age[count] >= 18 then add 1 to total** *//counts how many people are at least 18*

**end repeat**

### Nested loops

**repeat with classNumber = 1 to numberOfClasses**

**ask "How many students in class number “ & classNumber & “?"**

**if the result = "cancel" then exit to top**

**put it into numStudents**

**repeat until (numStudents >= 1) AND (numStudents <= 5)** //validates for 1 to 5 students

**answer "Invalid number - try again"**

**ask "How many students? "**

**if the result = "cancel" then exit to top**

**put it into numStudents**

**end repeat**

**.**

**.**

**.**

**end repeat**

## Testing

### Test data

The purpose of test data is to determine whether or not the program is **fit for purpose**

For example: to test input validation code for data in the range 30 to 60 inclusive

test data: **normal** values 45, 33, 57, … *// these should be accepted – in range*

**extreme** values 30 and 60 *//these should be accepted- in range*

**exceptional** values 29, 61, -0.008, “AAA” *//these should be rejected – out of range*

A test table is made to show the test results. For example, test data for validating integer data in the range 100 to 200 inclusive might look like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type of test data** | **Values used** | **Expected results** | **Actual results** |
| **Normal** | 150, 182, 101 | Value accepted by program | <<Screenshot showing the result when the program runs>> |
| **Extreme** | 100, 200 | Value accepted by program | <<Screenshot showing the result when the program runs>> |
| **Exceptional**  (out of range) | 99, 201, -500 | Value not accepted by program | <<Screenshot showing the result when the program runs>> |
| **Exceptional**  (wrong data type) | “#banana”, “£”, 0.045 | Value not accepted by program. System error? | <<Screenshot showing the result when the program runs>> |

### Errors

**Syntax errors** (e.g. spelling mistake) e.g. **pat 30 into length**

**Logic errors** (i.e. faulty logic by programmer) e.g. **put length + breadth into area**

**Runtime/execution errors** (errors that cause the program to fail while it is running)

Examples: a display field doesn’t exist, ***dividing by zero***, a file used by the program doesn’t exist

## Evaluation

### Fitness for purpose

Fitness for purpose means that the program meets all of the requirements of the analysis stage. The program would be tested to ensure that it carries out all of the requirements of the analysis stage without failing.

The test table results can be used as evidence for this. For example, you might say:

“My code needed to validate data in the range 100 to 200 and the test table shows that my code works for normal data. In order for the code to work for extreme data, I had to have the condition >= 100 not just >100. I fixed this logic error and my code now works for extreme data. My code does not work well for exceptional data – it rejects 99 and 201 which is correct – but I didn’t know how to have the code reject other types of data such as strings and reals. So my evaluation is that my program is reasonably fit for purpose but not completely”

### Efficient use of coding constructs

See *Efficient Solutions* in the Design section

This part of the evaluation would be where you judge whether or not you managed to write efficient code. Efficient code would involve using loops where possible, organising IF statements properly, using pre-defined functions, parameters and so on

For example,

**IF (x >=0) THEN**

**IF (y >= 0) THEN**

**IF (z >= 0) THEN**

**….**

**END IF**

**END IF**

**END IF**

would be more efficient written as

**IF (x >= 0) AND (y >= 0) AND (z >= 0) THEN**

**…**

**END IF**

This code is more *readable* (therefore easier to edit), it is more memory-efficient (*fewer lines of code*) and more processor-time efficient (processor has less work to do)

### Robustness

A robust program is one that can cope with errors during execution and with unusual input. It doesn’t fail (crash) when running.

For example, if a program needed to access another file to run and that file didn’t exist, a robust program would alert the user to the fact whereas a non-robust program would probably just crash.

If the program needed the printer to be online while it was running, a robust program would cope with the situation if the printer wasn’t connected – maybe alert the user and make suggestions.

Another situation that can make a program crash is trying to divide by zero.

When you are testing your programs, you should try different ways to make them crash to see how robust they actually are. Then you can make evaluative statements such as “it is robust for normal and extreme data but using “£” as exceptional test data made the program crash so my program is not completely robust”.

### Readability of code

It is very important to make your code readable, in case another programmer (or even yourself!) has to edit your code. Code can be made more readable by using:

* Meaningful variable names
* Internal commentary
* Indentation
* white space

Your evaluation should judge whether or not you have made your program readable and, if so, in what ways and that it meets all the requirements from the analysis stage.