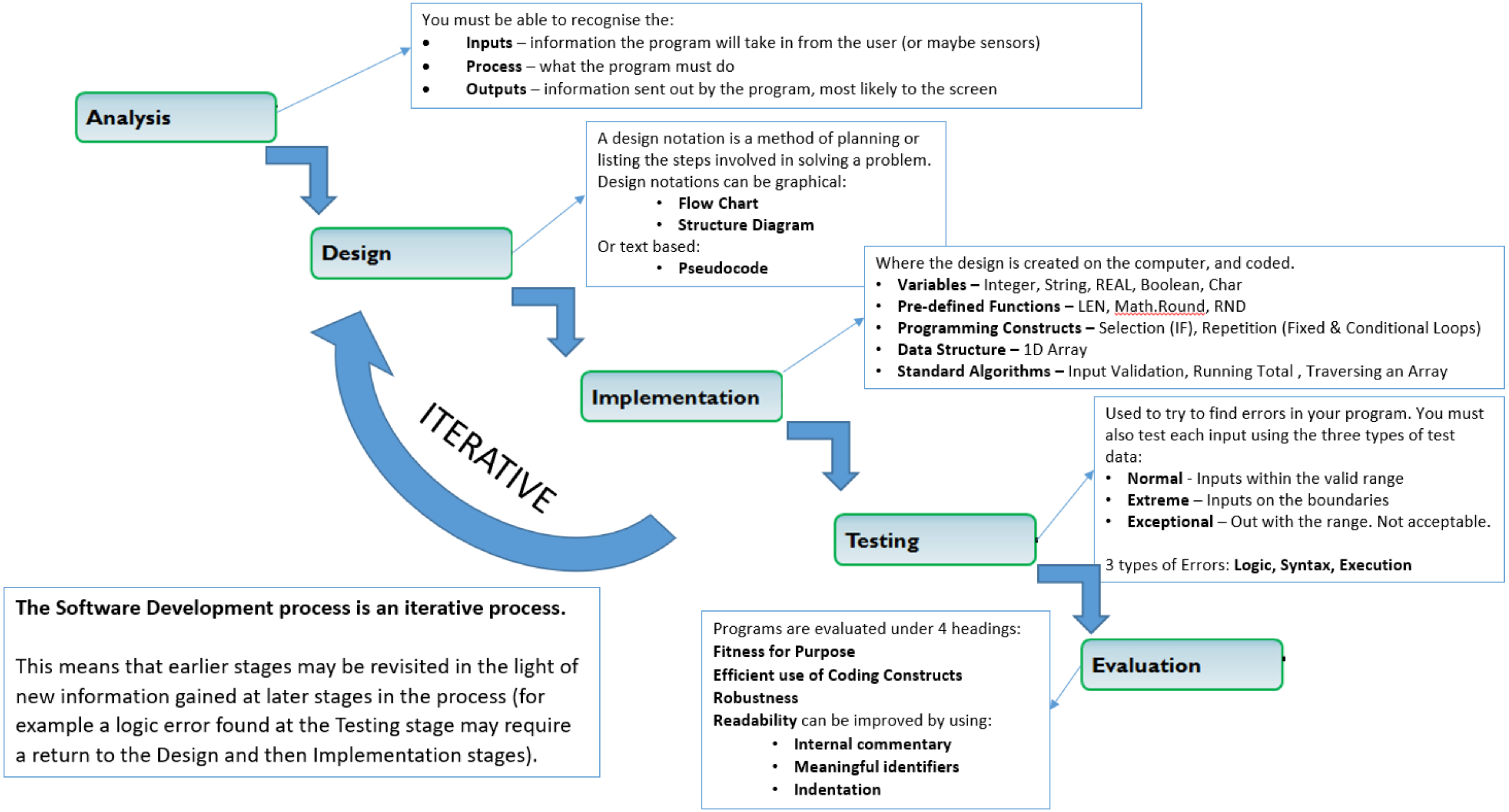
Software Design & Development – Summary notes

The Software Design & Development Process



You must be able to recognise the:

- **Inputs** – information the program will take in from the user (or maybe sensors)
- **Process** – what the program must do
- **Outputs** – information sent out by the program, most likely to the screen

A design notation is a method of planning or listing the steps involved in solving a problem.
Design notations can be graphical:

- **Flow Chart**
- **Structure Diagram**

Or text based:

- **Pseudocode**

Where the design is created on the computer, and coded.

- **Variables –** Integer, String, REAL, Boolean, Char
- **Pre-defined Functions –** LEN, Math.Round, RND
- **Programming Constructs –** Selection (IF), Repetition (Fixed & Conditional Loops)
- **Data Structure –** 1D Array
- **Standard Algorithms –** Input Validation, Running Total , Traversing an Array

Used to try to find errors in your program. You must also test each input using the three types of test data:

- **Normal** - Inputs within the valid range
- **Extreme** – Inputs on the boundaries
- **Exceptional** – Out with the range. Not acceptable.

3 types of Errors: **Logic, Syntax, Execution**

Programs are evaluated under 4 headings:
Fitness for Purpose
Efficient use of Coding Constructs
Robustness
Readability can be improved by using:

- **Internal commentary**
- **Meaningful identifiers**
- **Indentation**

The Software Development process is an iterative process.

This means that earlier stages may be revisited in the light of new information gained at later stages in the process (for example a logic error found at the Testing stage may require a return to the Design and then Implementation stages).

Software Design & Development

Describe and implement the phases of an iterative development process:

- **Analysis** – production of a software specification that will outline what the software must do
- **Design** – plan the design of the program, and the user interface
- **Implementation** – write the code for the program/ software
- **Testing** – fully test the program/ software to ensure that all aspect work correctly
- **Documentation** – technical guide – how to install software - and user guide – how to work the software.
- **Evaluation** – is the program fit for purpose and robust

This is an iterative process – each stage may have to be revisited as a result of new information coming to light.

Analysis

Identify the purpose and functional requirements of a problem in terms of:

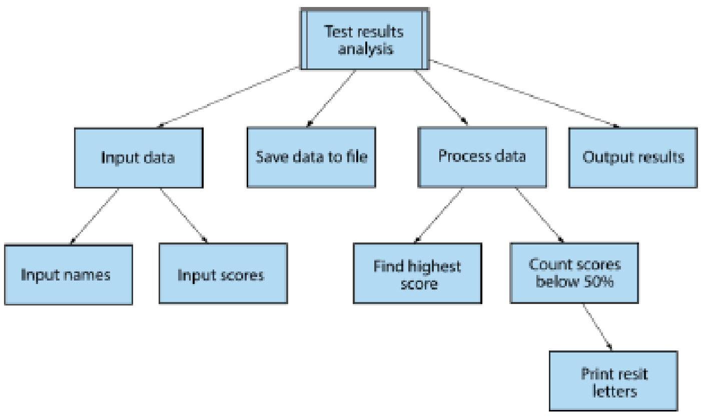
- **Inputs** – data that is entered into the program.
- **Processes** – what the program does with the inputted data.
- **Output** – data that is outputted to the user.

Design

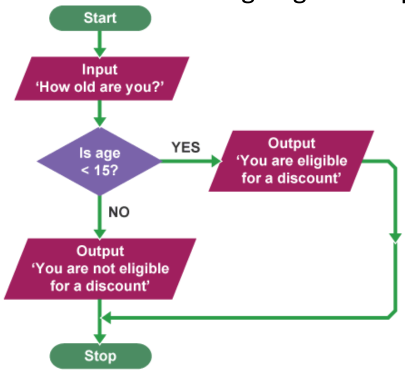
Identify the data types and structures required for a problem that relates to the implementation at this level, as listed below.

Describe, identify, and be able to read and understand:

- **Structure diagrams** – breaks the problems down into smaller sections.



- **Flowcharts** – shows what is going on in a program and how data flows around it.

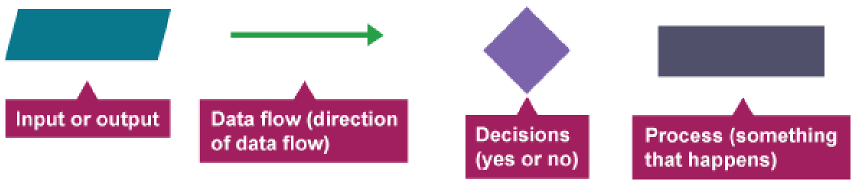

Start
Input 'How old are you?'
Is age < 15?
YES
Output 'You are eligible for a discount'
NO
Output 'You are not eligible for a discount'
Stop

Design

- **Pseudocode** – structured English for describing a program, read by a human

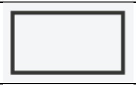
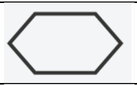
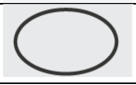
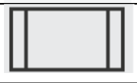
```
get age from user
if age < 15?
   Send discount to user
else
   Send no discount to user
end if
```

Exemplify and implement one of the above design techniques to design efficient solutions to a problem.
Below are the main flow chart symbols.



Input or output
Data flow (direction of data flow)
Decisions (yes or no)
Process (something that happens)

Below are the main structure diagram symbols.

	Process		Selection
	Loop		Pre-defined function

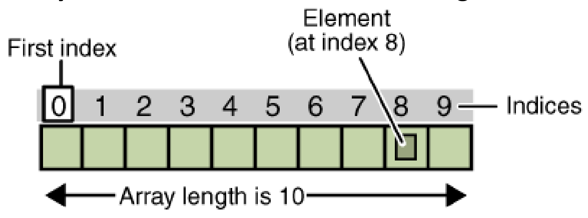
Describe, exemplify, and implement user-interface design, in terms of input and output, using a wireframe.
A wireframe is designed to provide a visual draft of how the interface could look. A wireframe could be anything from a rough paper-based sketch to a full and detailed image.

Implementation (Data Types & Structures)

Describe, exemplify, and implement appropriately the following:

- **Data types**
  - **string** – stores text and alphanumeric combinations e.g. Sausage, AB12 4RT
  - **character** – stores a single character which can be a letter, number or symbol e.g. Q. #, 6
  - **integer** – stores positive and negative whole numbers e.g. 17. -45. 124
  - **real** – stores numbers with decimal places, can also store integers e.g. 3.14, - 5.67
  - **boolean** – stores TRUE or FALSE only.

- **Data structures -**
  - **arrays** – stores a list of related data e.g. names of pupils in a class.



Implementation (Computational Constructs)

Describe, exemplify, and implement the appropriate constructs in a high-level language:

- **Assigning values to variables**.
  Variables are locations in memory where programs store and retrieve information.
  `SET score TO 5`
  `SET score TO score + 12`
  `SEND score TO DISPLAY` → 17 will be outputted to the user.

- **Arithmetic operations**
  - **addition** – `SET answer TO a + b`
  - **subtraction** – `SET answer TO a – b`
  - **multiplication** – `SET answer TO a * b`
  - **division** – `SET answer TO a – c`
  - **exponentiation** – *to the power off* **–** `SET answer TO a^b` *e.g. 2^3 = 8*

- **Concatenate strings -** concatenation is the process of joining two or more strings together to make a new text string. `SET displayName to firstName & surname.`

- **Selection -** The program selects an action dependant upon the value of variables held in the program
  - **If … Then** – `IF` *`condition`* `THEN` *`action`* `END IF`
  - **If … Then … Else** – `IF` *`condition`* `THEN` *`action_a`* `ELSE` *`action_b`* `END IF`
  - **If … Then … Else If** – `IF` *`condition_1`* `THEN` *`action_a`* `ELSE IF condition_2 THEN action_b END IF`

- **Conditional Statements**
  - **Simple Conditions** – depends on one conditional statement being true.

Operator	Explanation	Example
<	Less than	`age < 18`
>	More than	`age > 18`
<=	Less than or equal to	`age <= 18`
>=	More than or equal to	`age >= 18`
= or ==	Equality	`age == 18`
≠ or !=	Inequality/Not equal to	`age != 18`

  - **Complex Conditions**
    - **AND** – both conditions must be true e.g. `gender = "M" AND age > 18`
    - **OR** – either of the conditions must be true e.g. `class = 1 OR class = 3`
    - **NOT** – the condition should not be true e.g. `NOT (gender = "M")`

- **Iteration and repetition -** Iteration is the process where program repeat a group of instructions.

  - **Fixed Loop** – when a group of instructions is repeated a pre-determined number of times.

```
FOR counter 0 TO 5 DO
     SEND "I must behave" TO DISPLAY
End For
```

  - **Conditional Loop** – will keep repeating a group of instructions until a specific conditional is met.

```
GET password FROM (STRING) KEYBOARD
WHILE password != "letmein" DO
    SEND "In-correct" TO DISPLAY
    GET password FROM (STRING) KEYBOARD
END WHILE
```

Implementation (Computational Constructs) – cont

- **Predefined functions (with parameters):**
  A pre-defined function is a segment of code that can be referred to and it will do something useful.

  - **Round** – rounds a value to a specific number of decimal places.
    ```
    SET pi TO 13.141592
    SET pi TO ROUND (pi, 2)
    ```
    → *the new value of pi will be 13.14*

  - **Length** – give the number of character present in a variable.
    ```
    SET word TO "Mississippi"
    SET numWord TO LENGTH (word)
    ```
    → *the value of numWord would be 11*

  - **Random** – generates a random number, between a lower and upper range.
    ```
    SET bonusBall to RANDOM (1,59)
    ```

Implementation (Computational Constructs)

Describe, exemplify, and implement standard algorithms:

- **Input validation** - used to check that the input is acceptable.
  A program must ensure the user enters a number between 1 & 100.
  ```
  GET score FROM (INTEGER) KEYBOARD
  WHILE score < 1 or score > 100 DO
      SEND "In-Valid" TO DISPLAY
      GET score FROM (INTEGER) KEYBOARD
  END WHILE
  ```
  1. Get initial score from the user
  2. Check if data if valid, if invalid then enter loop
  3. Display error message
  4. Get new score from user
  5. Repeat loop until data is valid

- **Running total within loop**
  A program must calculate the total cost of 5 items.
  ```
  SET total TO 0
  FOR counter 0 TO 5
      GET itemPrice FROM (REAL) KEYBOARD
      SET total = total + itemPrice
  END FOR
  ```
  1. Set the initial total to 0
  2. Fixed loop, 5 times.
  3. Receive item price from the user.
  4. Add the item price to the total
  5. End Loop

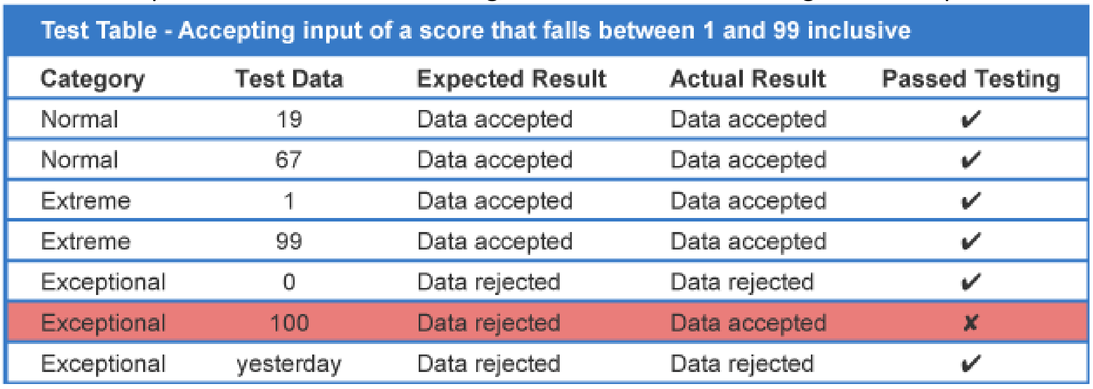
- **Traversing a 1-D array** – to transverse an array means to access each element stored in the array so that the data can be checked or used as part of a process.
  A program is required to count all A grades.
  ```
  SET grade TO [A, B, C, A, A, B, A, C]
  SET numA TO 0
  FOR counter 0 TO 7
      IF grade[counter] == "A" THEN
           numA = numA + 1
  END FOR
  ```
  1. Declare Array with all grades
  2. Number of A grades counted set to zero
  3. Fixed loop, loop for each item in array
  4.Check the grade at index if it is an A
  5. Add one to number of A grades counted
  6. End Loop

Testing

Describe, identify, exemplify, and implement normal, extreme, and exceptional test data.
The purpose of testing is to detect and remove errors in a program. Programs should be comprehensively tested to see whether they give the correct results when dealing with normal, extreme and exceptional test data.

- **Normal** – data that you would expect to work or be accepted (value within the range)
- **Extreme** – data at the lower and upper limits of what is expected (lowest value or highest value in range)
- **Exceptional** – data that should not be accepted by the program (value outside of the range)

- **Test Table** – provide a structure to testing, and evidence that testing has taken place.



Test Table - Accepting input of a score that falls between 1 and 99 inclusive

Category	Test Data	Expected Result	Actual Result	Passed Testing
Normal	19	Data accepted	Data accepted	✔
Normal	67	Data accepted	Data accepted	✔
Extreme	1	Data accepted	Data accepted	✔
Extreme	99	Data accepted	Data accepted	✔
Exceptional	0	Data rejected	Data rejected	✔
Exceptional	100	Data rejected	Data accepted	✘
Exceptional	yesterday	Data rejected	Data rejected	✔

As one test has failed, the programmer should now review the code to identify why this test has failed – and rectify the error. Screenshots can be used to provide evidence.

Describe and identify syntax, execution, and logic errors.

- **Syntax** – This is an error in the spelling or grammar used when coding.
  For example, misspelt command words, missing brackets, commands or colons.
  Syntax errors are identified by the interpreter, as the code will not run.

- **Execution** – This type of error occurs when the program is asked to do something that it cannot, resulting in a crash. For example, dividing by zero or trying to store a string a integer variable.

- **Logic** – This is an error in the logic of the code e.g. using a < instead of >. The program will run, but it will produce unexpected results. Logic errors are usually only resolved by carrying out testing.

Evaluation

Describe, identify, and exemplify the evaluation of a solution in terms of:

- **Fitness for purpose** – does the program do what it is supposed to do?

- **Efficient use of coding constructs** – a program is efficient if the length of the code is proportional to the scale of the project.
  - **use of loops** –repeating instructions is more efficient than a sequence of individual instructions.
  - **arrays** – using arrays to store related data is more efficient than using multiple variables.
  - **nested IFs** – using IF … ELSE IF … is more efficient that using multiple IF … THEN

- **Robustness** – can the program cope with unexpected input or mishaps without crashing?

- **Readability:**
  - **internal commentary** – add descriptions, notes and explanation to the code.
  - **meaningful identifiers** – use variable names that describe what the variable contains.
  - **indentation –** makes it easier to identify constructs. Mandatory in Python.
  - **white space** – help makes it clearer where code is placed.