Testing and documenting solutions

Testing should be **systematic**, **exhaustive** and **comprehensive** i.e. methodical with **test reports** of predicted and actual results kept, and tested under all operational situations with a full range of test data.

The three types of testing are **normal, extreme** and **exceptional.**

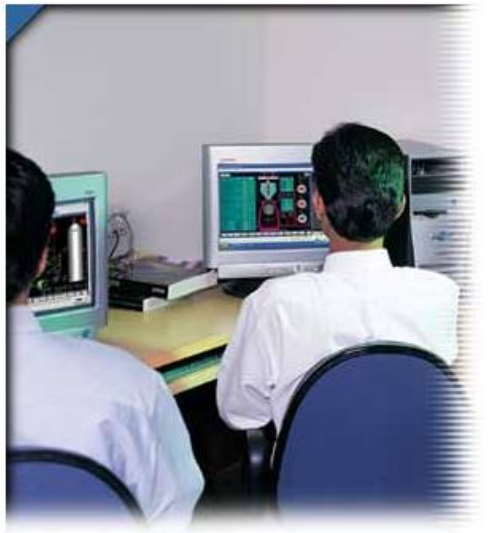
Example

Program should only accept whole values in the range 0 to 100:

Normal data: 2, 34, 66 etc.

Extreme data: 0,100

Exceptional: -1, 156, abc, 2.9 etc.



Usability (beta) testing is when **independent test groups** and/or the client try out the software and report back any bugs to the development team prior to final release.

Test plan

A test plan will include:

The software specification against which the results of the tests will be evaluated.

The **software specification** is produced at the end of the analysis stage of the software development process, and is a legally binding document which protects both client and developer. The design of the test plan must take this document into account.

A schedule for the testing process.

The testing schedule is necessary for the same reason as every other part of the software development process needs to scheduled in order to deliver the project on time.

Details of what is and what is not to be tested.

Exhaustive testing - where every possible input and permutations of input to a program are tested - is not possible. Even a simple input validation routine could theoretically need to be tested with every possible valid number, and the possibilities run into millions once you have several different inputs which could be applied in any order. The tests selected should be ones which are practical within the time available. There will always be external circumstances which cannot be tested until the software is in the hands of the client or the user base. This is where **acceptance testing (beta testing)** is important.

The test data and the expected results.

A test plan will include **normal**, **extreme** and **exceptional** test data; the results expected from inputting this data to the program and whether the result passes or fails the test.

Debugging and Errors

Debugging is the process of finding and correcting errors in code.

A **syntax error** is one which can be spotted by a translator: by a compiler when the source code is translated into machine code, or by an interpreter while the code is being entered by the programmer.

Examples: misspelling of a keyword, or a mistake in the structure of a program like a missing END WHILE in a WHILE loop or a missing END IF in an IF condition.

An **execution error** is one which happens when the program is run, causing it to stop running (crash). Examples include division by zero or trying to access an array index that's beyond the range of that array. These types of error are not identified by the compiler or the interpreter, but appear when the program is run.

Logical errors, sometimes called **semantic errors**, are ones where the code is grammatically correct as far as the interpreter or compiler is concerned, but does not do what the programmer intended. These types of error may be spotted during the implementation stage, but may also be spotted during the testing stage.

Debugging Tools

Dry runs

A dry run is simply a manual run-through the pseudocode or source code of the program, usually taking notes of the values of variables at various points in the process while doing so. In effect the person doing the dry run is taking the place of the computer in order to check that the code is doing what they expect it to do. Keeping track of the values of variables at different stages of the code execution is complicated so normally the tester would use a table, either on paper or on computer to help.

Debugging Tools

Trace tables

A trace table is similar to the table that would be used during a dry run, but is often used to test an algorithm for a specific sub program when the tester wants to check the result of a number of different values of a variable. A trace table and its results is an important element in the documentation of the testing process.

```
SET numbers TO [3, 15, 4, 7, 8]
PROCEDURE findMaximum(numbers)
SET maximumValue TO numbers[0]
FOR counter FROM 0 TO 4 DO
    IF maximumValue < numbers[counter] THEN
        SET maximumValue TO numbers[counter]
    END IF
END FOR
SEND [“The largest value was “& (STRING) maximumValue] TO DISPLAY
END PROCEDURE
```

Line no	numbers[0]	numbers[1]	numbers[2]	numbers[3]	numbers[4]	maximumValue	counter	number[counter]
1	3	15	4	7	8			
2								
3						3		
4							0	
5								3
6						15		
7								
8								
4							1	
5								15
7								
8								
4							2	
5								4
7								
8								
4							3	
5								7
7								
8								
4							4	
5								8
7								
8								
9								
10								

Trace Tools

Breakpoints

Some programming environments will enable the programmer to set a **breakpoint**. Setting a breakpoint in a program sets a point in the source code where the program will stop execution, at which point the values of variables at this point can be examined. Breakpoints can be set to stop execution at a particular point in code, or to stop when a variable has a particular value (watch) or a particular key is pressed. Once the program has stopped, the values of the variables in use can be examined, or written to a file for study later.

Watchpoints

Whereas a breakpoint is a specific place in a program where you want it to stop, a watchpoint is where you set a program to stop when a variable has a specific value or when a particular event such as a keypress, data entry or a menu selection has occurred. When using either a breakpoint or a watchpoint, the purpose is to track the flow of data or the changes in values of variables in order to debug the code.