Testing & Review

Testing & Debugging
Testing is the part of the **quality assurance** process in the software development life cycle. Thorough testing **guarantees** that the solution should **work properly** and **meet** the **identified needs** of the **stakeholders** (as detailed in the : req**uirements specification).**

As a general rule, **good** programs are RARE!

Having a thorough **testing strategy** is the key to creating RARE programs. This starts with the concept of a **test plan.**

RARE programs are:
Robust - they do not crash when given bad or silly input.

Accurate - they produce results with an acceptable level of accuracy.

Reliable - they work the same way every time they are used; no unexpected surprises.

Efficient - they calculate results and perform operations as quickly as possibly.

Test Plan

A Test Plan should attempt to:

- Check different **logical pathways,** e.g. does the test check **both halves** (true and false) of a c**onditional statement (e.g.** an if. .. else)
- Check all **possible outcomes** of a **case** statement
- check all the loops been successfully tested?
  Flowcharts are particularly good for this as the **logical pathways** are easy to see. The tests should touch upon all **flow arrows** on the diagram.
- Check **normal data.** The program needs to be tested with data that is **within a sensible range;** data likely to be input.
- Check **extreme data.** It should also be tested with values, which while still within a sensible range are **less likely** to be input; these are the values on the **extremes** (both **high** and **low).** For example, age over 115 years is not impossible, but it is **extreme** (1 in 2.1 billion approximately!)
- Check **erroneous data.** Not all data entered into a program **will** be sensible. In order to ensure it robust, **spurious** data should be tested. This typically included **values outside valid ranges** and **wrong types of value (e.g.** alphabetic when a number is expected).

Collecting suitable test data is important, both in **quantity** and quality of **spread. The test plan structure should include:**

- **Test - what** is being tested; which **part** of the program is being tested.
- **Date - when** the test is taking place; this is important as it may link to a particular version of the program.
- **Expected result -** what results **we** expected to get out of the program by **tracing through first** on paper.
- **Actual result -** the results generated by the computer using our supplied test data.
- **Corrective action -** if a problem was discovered, what was done to the program to fix it.

Many formats exist for this type of content. A **trace table** is often seen as a simple way of **tabulating** and **comparing** such results.

Trace Table for program : Water Mover
Test: Calculation of water weight based on volume

Number	Box H	Box W	Box L	cubed expected	cubed actual	Calculated weight expected kg	Calculated weigh tact g	Corrective Action
1	10	20	30	6000	6000.0	0.001 x 6000 = 6kg	6.0	None
2	20	40	60	48,000	48,000.0	0.001 x 48,000 = 48kg	12.0	Fix data type problem

The trace table allows us to record the values **entered** and **logical pathways used** when a program runs both on paper (the `dry run') or live.
The **comparison** between the **actual** and **expected results** will quickly show how **accurate** the programmed solution is and also, given the variables involved in any particularly test, where any possible problem will be found.

Screen captures are a good addition to any trace table as they **reinforce** the **actual results** of the program running.

Error messages

The process of **finding** and **removing** errors from program code is known as **`debugging'.**

Some errors occur at **compile-time,** found as the high-level language is translated into low-level language. Statements which break the language's syntax generate **compilation errors.**

when errors occur while the programming is running, they are refe[r]red to as **run-time errors.**

Warnings are minor issues ,covered during translation; they , not fatal (like an error) but may indicate possible run-time errors. A common example is using a data type which is too small to hold a calculated result. The run-time effect would be truncation and, therefore, accurate results.

Specialist software tools

Modern programming software has feature-rich tools to assist the specialist while debugging a solution. typical IDE, the three most commonly used **debug tools** are the following.

- Trace

  A trace allows the programmer to **follow** a program **line by line** as it executes, walking through the different **logical pathways** as the program progresses.
  Tracing is *very* useful when **conditional statements (if...** else, case etc) are *present.* If the **trace** shows unexpected **behaviour** (going down the wrong logical pathway, for example) the programmer will need to check **their logic** to see where things have gone wrong.

- **Watch**

  A watch lets the programmer spy on the **contents** of a variable while the program is running, usually ; **a trace.** One of the most common programming problems is a variable storing **unexpected values.** The watch lets the programmer **see the changes in a variable's contents** as different lines of the program are executed.
  The apearance of an unexpected value in a watched variable will allow the programmer to **narrow their search** to just a few lines of program code.

- **Breakpoint**

  A Breakpoint is a debugging feature which lets the programmer **mark a line of code** with a physical breakpoint. When the program runs it will **halt temporarily** at this point.

  From here the programmer can decide to **trace the remaining code** and/or **inspect variable watches** they have set. The clear advantage here is that parts of the program that are functioning correctly **need not be traced;** the breakpoint can be placed after these sections have finished.

User Documentation

Unlike **technical documentation,** this type of documentation is meant to be read by a **typical end-**user - **not** a developer.
As a result, the user instructions should **avoid the use of technical terms or jargon** where possible.
Typical content may include:

- how to **install** the program (including `loading' instructions and hardware/operating-system requirements)
- how to safely **uninstall** the program
- how to **start** and **end** the program
- how to **use** the program **properly**
- **how to resolve** problems that might occur (also known as **a`troubleshooting'** guide)
- how to get **further help** (online forums, files on disk, telephone support etc.).

A common tactic is for the user guide to hold the user's hand by taking them through a typical example of the program working.
In addition, keeping the instructions brief **(step-by-step** are ideal), with accurate **screen, mouse** or **keyboard** diagrams or **screen captures** to show the program running.

Other forms of Documentation

Text files (e.g. Readme.txt)
An ASCII (American Standard Code for Information Interchange) text file containing basic information about the program. This is usually stored as a file on a disk or in a downloaded archive (e.g. ZIP or .RAR). Text files can be opened and created by using a text editor such as Microsoft Windows **Notepad.**
.PDF document
Adobe's portable document format file - a popular, secure and reliable way of sharing electronic documents. A copy of the freely downloadable Adobe Acrobat Reader® application is required inorder to read a.PDF file.
Screencast videos and animations
Why describe how something works when you can show it? Applications such as Wink® and AdobeCaptivate can be used to create recordings of programs being used. These recorded tutorials ('screencasts') often have additional highlighting and narration to help explain what is being seen onscreen.

System Review

A formal, reflective procedure that should clearly identify:

- the program's **strengths**
- the program's **weaknesses**
- **how well** the program **meets the stakeholders' needs.**

This is essentially a comparison of the **final product** and the **requirements specification.**

It will require the **cooperation** of all the stakeholders in order to secure the necessary findings and could take some time.
End-users of the software can be **interviewed, observed** or be given a **questionnaire** to complete. The use of **focus groups** to **dig down** into **user viewpoint** is also valuable. In addition it is likely that a section on further developments will be required, listing:

- an overview of **corrections** made (to errors)
- **improvements** (to existing performance) that could be made
- **expansions** (to existing functionality) that could be invested in.

Review should be an **ongoing process,** exploring program **performance** and **end-user satisfaction** at **regular intervals.** If faults are found which cannot be remedied using minor maintenance, the software development life cycle (as seen in 2.1) will start once more.

Exercise Testing

Name :

1) Name four qualities a good program should possess.

2) Name four properties that a test strategy should cover.

3) Explain the difference between an error and a warning

4) Name three specialist tools which assist the debugging process.

5) Name four different aspects a user guise should cover.

6) Name three different forms of documentation media.

7) Which three points should a formal program review identify?

8) What further developments can be identified during review.

9) What should user documentation avoid?