Debugging methods

1. Correcting Algorithms

An essential skill to learn as a computer programmer is the ability to spot mistakes in a program.

This process of spotting and correcting mistakes is called 'debugging'.

This section has been written assuming that your algorithm is described using pseudocode, but most of the techniques will work equally well if you have used a flowchart.

Pseudocode algorithms are read from top to bottom, with one statement per line. The first step to correcting an algorithm is to read it through carefully and understand what each statement is meant to be doing.
Only once you understand what the code is *meant to be doing* can you move forward to correcting it.

Debugging methods

2. Errors in logic

Look at the following sentence:

My pencil just rode a bike to London

While it is a grammatical sentence, it makes no sense. Pencils don't ride bikes. Somewhere, the programmer writing this sentence has made a *logical error.* Only the programmer can spot these logic errors because the computer will just do exactly as it has been told.

Here is a very common logic error.

Infinite loops

This pseudocode has a logic error

```
Line 30:          mynumber = 1
Line 31:            WHILE mynumber != 2
Line 32:              PRINT mynumber
Line 33:            END WHILE
```

The mynumber variable is set to 1 in line 30, and nothing ever changes it - so the code loops around forever. The code below guarantees the while loop can end because of the increment mynumber command at line 33

```
Line 30:          mynumber = 1
Line 31:            WHILE mynumber != 2
Line 32:                PRINT mynumber
Line 33:                INCREMENT mynumber
Line 34:              END WHILE
```

Debugging methods

3. Arithmetic order errors

Not using arithmetic order properly

In maths, you should be familiar with the *order* of operations when working out a calculation (BODMAS)

```
20 + (1+5)/2 - 3*4
```

First work out the brackets, then multiply and divide, then add or subtract.

Let's look at some broken code (below). This code is to work out miles per gallon. It takes the starting and end mileage of the car and the amount of petrol used.

```
Line 30:            startMiles = 20
Line 31:            endMiles = 26
Line 32:            petrolUsed = 2
Line 33:            milesPerGallon = EndMiles - StartMiles/petrolUsed
```

In line 33, by slotting in the value of each variable, the miles per gallon value is going to be worked out as

milesPerGallon = 26 - 20/2

The answer produced by this code is 16, which is clearly wrong. This is because the order rules of maths say always divide before subtracting. The correct code in line 33 is

```
Line 33: milesPerGallon = (EndMiles - StartMiles)/petrolUsed
```

The brackets have been added to force it to work as intended.

Debugging methods

4. Logic errors cont.

Loop is off by one

This is a very common programming mistake - the code loops around one less than intended.

```
Line 30:            COUNT = 3
Line 31:            FOR i = 1, i < COUNT
Line 32:                print i
Line 33:            NEXT i
```

The intention is to print out three values of i, namely 1,2,3. Can you see why it prints out 1,2 but not 3?

The problem is with i < COUNT, because the FOR loop ends when i becomes 3 and does not print out the last i. To fix it the compare has to be i <= COUNT i.e less than OR equal to COUNT. This is the corrected code

```
Line 30:            COUNT = 3
Line 31:            FOR i = 1, i <= COUNT
Line 32:                print i
Line 33:            NEXT i
```

Forgetting that Arrays begin at position zero and not 1

The first item in an array is at index 0 in most programming languages and not 1. Forgetting this fact can lead to some odd results.

Let's write some broken pseudocode to print out the first value of an array

```
Line 30:            MyArray = [5,6,7,8]
Line 31:            PRINT MyArray[1]
```

This prints "6" instead of "5"!

The correct code to print out the first value would be:

```
Line 30:            MyArray = [5,6,7,8]
Line 31:            PRINT MyArray[0]
```

Debugging methods

5. Syntax errors

While logic errors can be picked up by looking at the pseudocode for an algorithm, certain types of errors are only introduced when the algorithm is written out as actual source code. Some of the most common mistakes are 'syntax errors', caused by using incorrect grammar for that particular language.

Can you spot the syntax error in the English sentence below?

"I is able to spot an error"

Of course the 'is' should be 'am' because we understand the syntax rules of English.

Like actual human languages, every programming language has its own set of 'grammar' rules. These rules set out the correct way of putting the code together. Since computers are very literal, if you get the syntax wrong, they will usually fail to understand what you were trying to do.

Some of the most common syntax mistakes are itemised over the next few pages. Note, they are not programming language specific.

Debugging methods

6. Syntax errors

Missing end of line marker

Algorithms are written with one instruction per line. Many programming languages indicate the end of a line using a particular marker, such as a semi-colon.

In this type of language, the command

```
                 Line 30:            PRINT result
```

has a syntax error because the semi-colon ; is missing.

```
                 Line 30:            PRINT result;
```

Missing bracket

Certain sections of code, like the contents of a list, are identified by putting them between brackets. If you forget to close those brackets, the program doesn't know where to stop looking for items to put in the list.

The syntax rule for this language is '*a list opens with a square bracket and ends with a closing bracket*'

```
                 Line 10:            MyList = [1, 4, 6, 15
```

To fix this syntax error, the code would have to be changed to

```
                 Line 10:            MyList = [1, 4, 6, 15 ]
```

Curly brackets are often used in a program structure called a procedure or subroutine to mark their beginning and end.

Like this

```
                        procedure MyProcedure () {
                            ... some code
                            .... some more code



                       }
```

Forgetting to add the closing bracket is easily done and will always lead to an error.

Debugging methods

7. Syntax errors cont.

Misspellings

I'm sure you've sent e-mails or texts where a word is misspelled. Unfortunately, while a human can generally work out what you were trying to say, a computer is more literal.

```
Line 30:            MyFruit = "Apple"
Line 31:            JohnsFruit = "Pear"
Line 32:            PRINT MyFriut
Line 33:            PRINT JohnsFruit
```

The syntax error here is on Line 32. Instead of printing out MyFruit, it tries to print out MyFriut. The correct code would be:

```
Line 30:            MyFruit = "Apple"
Line 31:            JohnsFruit = "Pear"
Line 32:            PRINT MyFruit
Line 33:            PRINT JohnsFruit
```

Case Sensitive errors

Some languages are case sensitive, so even a mistake between upper and lower case is a problem. For example:

```
Line 30:            MyFruit = "Apple"
Line 31:            JohnsFruit = "Pear"
Line 32:            PRINT myfruit
Line 33:            PRINT johnsfruit
```

Lines 32 and 33 both contain syntax errors, because what they're asked to print is not the same upper and lower case as what we want it to print.

```
Line 30:            MyFruit = "Apple"
Line 31:            JohnsFruit = "Pear"
Line 32:            PRINT MyFruit
Line 33:            PRINT JohnsFruit
```

Debugging methods

8. Syntax errors cont.

Missing end of string marker

Programs handle many different types of data. One of these types of data is called a 'string', which is just a programming name for a piece of written text. To tell strings apart from the rest of the code, a very common syntax rule is '*all text strings start with " and end with "*'.

```
Line 30:            print ("This is a syntax mistake);
```

If you look carefully there is no " at the end of 'mistake'

```
Line 30:            print ("This is a syntax mistake");
```

Other syntax mistakes

Other common syntax mistakes are

- No return statement in a procedure - forgetting to actually return the result
- Mixing up left and right in an = statement. e.g. result = a + b is correct but a + b = result is not.
- Incompatible data type. e.g. if 'Score' is a byte variable, then Score = 5000 is a mistake.
- Using = when == should have been used e.g. IF (num = 100) is wrong but IF (num == 100)is correct
- Not closing WHILE and FOR loops properly such as missing the NEXT statement

Debugging methods

9. Debugging tools and methods

As you can see, there are many easy ways to make errors that will cause your algorithm or source code to stop working. It takes practice to be able to identify them all and to identify them quickly.

However, there are tools and methods available to spot errors that are a bit more scientific than just staring at the code hoping for inspiration. These include:

- Comments
- Debugging outputs
- Break points
- Single-stepping
- Test code

Make extensive use of comments

Adding comments to your code makes it much, much easier to understand later on. For example the // lines are comments

```
  // Calculate the total price including tax

        TotalPrice = price * 1.2

  // Send the information to the database and return if its successful or
not
       result = UpdateDatabase(TotalPrice)
```

Those two extra lines of comments allows someone to read the code and understand what it is *supposed* to be doing. Then it is easier to spot a logic error if it is misbehaving.

Debugging methods

10. Debugging tools and methods cont.

Place debugging output statements

An excellent way to see what the code is doing is to place temporary output commands at key points in the code to display what the value of a variable is at that particular moment in the algorithm.

The output is usually displayed directly on screen or perhaps in some special 'output' window.

For example, you may be writing some loop code and want to see how a variable is changing

```
Line 30:        COUNT = 3
Line 31:        FOR i = 1, i < COUNT
Line 32:              ...... some code
                  output i    // debug code
Line 33:        NEXT i
```

The `output i` statement has been placed to temporarily display the value of i as it loops around. If you are happy with what it is doing, then it is either commented out in the final code or removed altogether - like this

```
Line 30:        COUNT = 3
Line 31:        FOR i = 1, i < COUNT
Line 32:              ...... some code
                  // comment line output i
Line 33:        NEXT i
```

Debugging methods

11. Debugging tools and methods cont.

Set break points and single stepping

Many programming languages are available as a complete development package that includes a code editor, complier and debugger built into one application. It is called an IDE, an Integrated Development Environment. A key feature of an IDE is a built-in debugging tool.

A 'break point' is a command for the debugger to stop running the code right at that point. Then you can examine the state of all the variables to check that they are correct. If they are not correct, then stopping the code in this way lets you understand how and what set those particular values.

```
Line 30:        num = 3
Line 31:        myval = 2
Line 32:     ->   myval = myval + num
Line 33:        c = 3
```

A breakpoint in the code is usually shown as a symbol next to the line of code, the one above uses -> to indicate a break point.

Another feature usually available in a debugger is to 'single step' through the code, line by line and view the output and variables as you do so. Every time you press the single step key, another line is run and then it stops.

Debugging methods

12. Debugging tools and methods cont.

Write test code

If part of the code is misbehaving, a good way to spot the logic error is to write some test code and apply it to that chunk of code.

For example, say a loop is going to store a very large file.

```
Line 30:        myVeryLargeFile (1000)
Line 31:        for i = 1 to i < myVeryLargeFile.length - 1
Line 32:            store myVeryLargeFile[i]
Line 33:        next i
```

There seems to be problem as not all the file is stored - some parts are missing. Rather than deal with the large file, let's swap it for a much smaller one and write some test code. So now you insert some test code like this

```
Line 30:  // myVeryLargeFile (1000)  commented out whilst debugging
Line 31:  myVeryLargeFile = [4,5,6] // test code with a small data set
Line 31:    for i = 1 to i < myVeryLargeFile.length -1
Line 32:      store myVeryLargeFile[i]
Line 33:     output i  // debug code
Line 34:     output myVeryLargeFile[i]   // debug code
Line 33:    next i
```

The proper large data set on line 30 has been commented out with // so it no longer does anything, then a tiny version with the same name is inserted as a piece of test code `myVeryLargeFile = [4,5,6]` in line 31. In addition, an `output i` command is added to the code and an output myVeryLargeFile[i] which lets you monitor what is happening as it loops. Lo and behold, you see that it is only outputting 5,6 and missing the first item, 4. The problem is the classic mistake of starting i at 1 and not 0.

Now you fix the code and remove the test code as its job is done.

Debugging methods

13. Run Time errors

The errors so far, both logic and syntax, can be found fairly easily by reading through the source code you've written.

There is one other type of error that is more difficult to find, called a **run time error.** Run time errors only show up as the program is actually running. There are three common types of run time error:

Overflow error

An 'overflow' occurs when a value is too big to be stored in the memory the program has assigned to hold it. This happens because all data types have a maximum value they can hold.

For example, the maximum value of an unsigned byte variable is 255.

```
SET Num1 TO (BYTE) 254
SET Num2 TO (BYTE) 2
SET TotalNum TO (BYTE) Num1 + Num 2
```

The last line where Num1 and Num2 are being added to result will cause an overflow error because the result is 256 and a byte cannot store this value.

Another fault that is guaranteed to cause an overflow error is to try to divide a number by zero.

Overflow errors will either crash the program immediately or produce unexpected results. The programmer will then have to determine why and under what conditions the overflow occured.

Stack Overflow error

When the program calls a function, it keeps track of where it needs to return once the function ends. It does this using a temporary store called the 'stack'. The stack only has so much available space, though. So if a program tries calling too many functions at once, the stack will get too full and the program won't know where to return to.

This is called a 'stack overflow' and once again, results in a crashed program.

When a stack overflow occurs, the programmer has to track down what caused it. This can get difficult with more complicated programs, but is often due to a function trying to call itself too many times.

Calling an absent library procedure

A program may use a subroutine from an external code library, for example a DLL library. If that DLL file is missing altogether, or the library subroutine itself is faulty, then this will cause a run-time error in the program.

Solving run time errors

To help pin down the cause of a run-time error, some development environments provide a real-time emulation and trace facility where the program is allowed to run within the test environment and can be stopped the instant the error happens. The programmer then traces back through the program states to find the fault.

Debugging methods

14. Summary

- Algorithms and programs involve following steps in sequence, one after another
- It is quite common to make mistakes while writing out an algorithm
- Logical errors are where the algorithm is doing something that doesn't make sense or not as intended
- Syntax errors are where the algorithm or program is written using incorrect grammar for the programming language.
- Common logical errors include accidentally making infinite loops, calculating things in the wrong order, or forgetting the correct position of an item.
- Common syntax errors include misspelling things, missing out characters, or using the incorrect letter case
- A run-time error occurs whilst the program is actually running
- Run-time errors include: Overflow error, Stack overflow error and missing external library.
- There are processes you can follow to make finding mistakes easier.
- These processes include:
    - adding comments describing the intended purpose of a line in an algorithm or program
    - checking the outputs at various stages before the program completes
    - telling the program to end early so you can check where exactly the error occurs.

Testing

1. Testing

Testing is an essential part of software development.

Testing ensures that the program

- Works correctly without error
- Meets the needs of the user

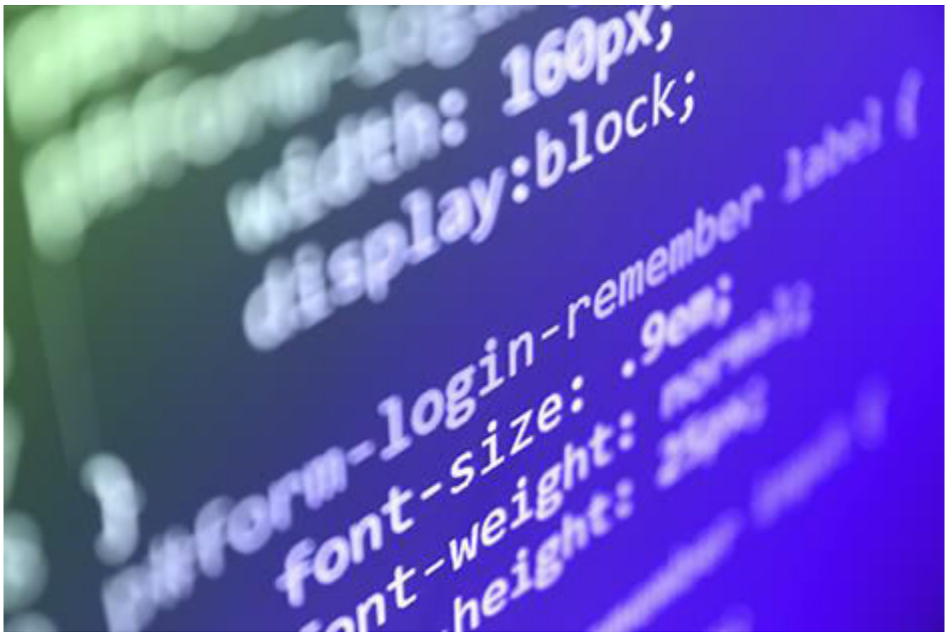
Testing needs to be planned *ahead* of actually writing the code. It also needs to be approached in a disciplined way.

This section will describe the two types of testing - iterative and final testing. It will also consider good testing methods.

Testing

2. Test plan

For personal projects, coders tend to just have an idea in their head of what the program needs to do when they start to actually write it. Then maybe along the way, they test small segments of code to make sure they're working correctly.



This is fine for small, personal programs that have a specific purpose. But for larger, professional-level projects there needs to be a test plan in place that is based on what the end-user needs.

What is a test plan?

First of all, there needs to be an understanding of what the user wants from the system. This is initially defined in the project specification and agreed with the user. Once the project outcomes have been clearly defined, the test plan can be written.

A test plan is a formal document which details the tests to be performed on the software. It describes:

- The scope of the testing
- The tests to be performed
- The reason for each test
- The data to be used in tests
- The expected outcome of each test

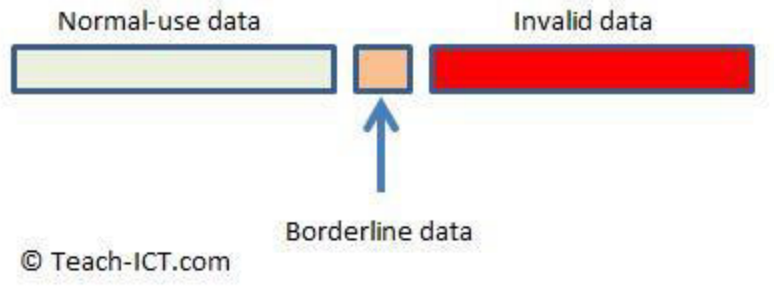
Once the test has been conducted, the actual result from the test, along with evidence (e.g. screenshots) is added to the plan.

Testing

3. Test data

Whilst writing the test plan, the 'test data' that is going to be used for each test, must be identified.

There are three types of test data :



1. Normal use data

This is the data that is *expected* to be entered into the application.
For example there may be an input form asking for a username. It is *expected* that the user will type in some letters and numbers for this. So normal use data will consist of usernames made of different combinations of letters and numbers.

The program needs to run without any errors when presented with this kind of data.

2. Boundary / Extreme data

This is testing the very boundary of acceptable data. Boundry data is still acceptable and it will be processed in the same way as normal data.

For example, the user is asked to enter a username with between 1 and 10 characters. The borderline test data would be a username with 1 character and a username with 10 characters.

Boundary data is excellent for testing the hard limits written into the software and that the application still runs properly when handling it.

3. Invalid or erroneous data

This is data that the program rejects as invalid. That might be because it is the wrong data type. It might be because it contains characters that are not allowed. Or it may be that the value falls outside the accepted parameters of the program.

For example, if the user is asked to enter a username with between 1 and 10 characters, invalid test data would be a blank username, or a username with 13 characters, or one that doesn't use standard letters and numbers (emojis or foreign characters, for example).

If invalid data is presented, then the application needs to handle it properly i.e. not crash. Usually the user is told that the data provided has been rejected.

Testing

4. Test Strategy - black box

With a "black box" testing strategy, the testers ignore how the program itself works in terms of code. They simply enter inputs and check if the outputs match their expectations.

The advantage of black box testing is that the person carrying out the test doesn't need to know anything about the software.

This makes finding it a lot easier to find testers, as you can pull them from a general pool or programmers or non-experts who have never seen the program before.

When you have a lot of test data to go through, black box testing allows you to check whether the program works very quickly and easily.
The disadvantage of black box testing is that if the test fails, the tester doesn't know *why* it failed.

Testing

5. Test Strategy - White box

A 'white box' approach looks into the details of every algorithm in the software. White box strategy involves **'unit testing'**.

Good coding practice encourages programmers to write their code into fairly independent chunks called an unit. Usually in the form of modules, subroutines, functions and procedures. The unit test seeks to test every combination of data that the unit is expected to deal with.

The unit test is usually carried out by the programmer but sometimes it is done by an independent person as well in highly critical software such as onboard a space probe where a fault can make or break a mission.

The white box approach tries to test every possible path through that bit of code and ensures that it works correctly every time.

One way this can be done is by first setting up the test input data and then "single-step" through the code by hand. The results of this are recorded in a "trace table". Let's use a simple loop as an example

```
START
a = 3
b = 2
WHILE a < 10
  c = a - b
  PRINT (c)
  a = a + 2
END WHILE
END
```

A trace table for this would show the results of every iteration of the loop, keeping track of the values of each of the variables. It would also leave space for comments, to help anyone who reads the table later. Like so:

Trace table				
Iteration of while loop	**Input (a)**	**Input (b)**	**Output (c)**	**Comment**
1	3	2	1	
2	5	2	3	
3	7	2	5	
4	9	2	7	
5	11	2		Since a is not < 10 the loop exits with the

Trace table				
Iteration of while loop	**Input (a)**	**Input (b)**	**Output (c)**	**Comment**
				values a and b

The advantage of white box testing is that if a test fails, you can quickly pinpoint exactly where the problem is and fix it.

The disadvantage of white box testing is that it can take a long time to run through all of the test data. You would have to make trace tables for all of the normal test data, and all of the borderline test data, and all of the invalid test data.

Testing

6. Black box vs white box testing

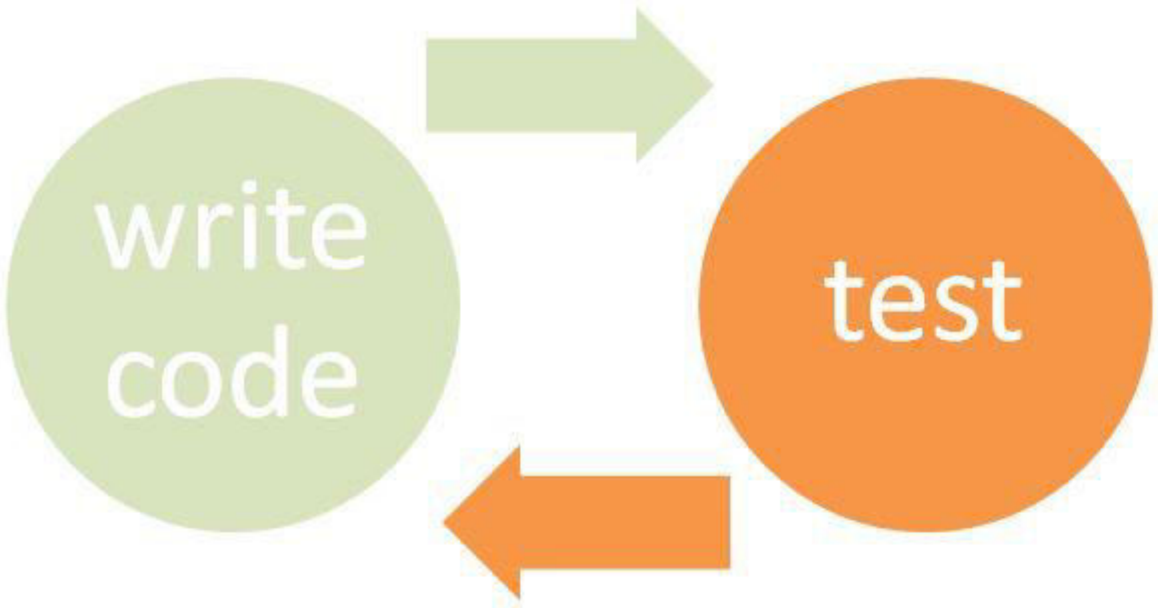
Successful Test	
Black box testing	**White box testing**
Fast and easy for the tester.	Much slower to complete tests compared to black box testing.
No wasted time, the test is completed and the tester can move onto the next one in the plan.	Some tests can be complicated and require knowledgeable testers
	You get a detailed record of the correct data, but often people are only interested in **whether** a program works, rather than **why** it works.

Failed Test	
Black box testing	**White box testing**
Provides no insight into why the test failed, unless you can work out a pattern from which test data led to failures and which to successes	By looking at the trace tables, you can see exactly where the failure occurred and quickly fix it

Testing

7. Iterative testing

During program development, the programmer will write a portion of their code. They will then test the code they have just written to check that it is working as expected before moving onto developing the next part, and testing again.
This is called **iterative testing.**



Iterative testing can be done at any level. You can iteratively test each line of code as you write it. Or you can test each *block* of code as you write it, checking that a loop, function or procedure does what it's supposed to do. Or if the project requires several programs, you can iteratively test each program as you write it.

Iterative testing is looking for two types of error:

- Syntax errors - mistakes in the grammar of the coding language
- Logic errors - mistakes in the algorithms or logic flow of the program

You can learn more about these types of errors and how they are picked up during testing in our section on **Types of Error**.
Iterative testing, as a general rule, is done using the "white box" testing method.

Testing

8. Final testing

Final testing, otherwise known as 'User Acceptance Testing', 'Terminal testing', or 'Project sign off', is done after the project is thought to be complete.

Final testing is carried out using the "black box" testing approach.

Just like the earlier tests, the process of final testing will have been laid out in the initial test plan, according to user requirements, complete with the exact test data to use and the expected outcomes.

Alpha and Beta testing

Final testing is often broken down into two stages: Alpha and Beta.

Alpha testing is when the software is released for use by in-house test engineers. They try their best to break it by doing unexpected things to it.

If it passes alpha testing, then the project goes to beta testing. This is where the software is released to a select group of hired or volunteer testers, unconnected to the development process to date.

These "beta testers" use the program as intended and under realistic conditions. Each beta tester reports back any faults or issues they find to the development team. The developers will try to track down the causes of these issues, fix them, and provide the beta testers with an updated version of the program to test again.

Release

Once the beta testing stage is finished, the testing plan (which now includes the results of all of the tests) is handed back to the user to check that the project does what they want it to do. If it does, then the code is finally released as a finished product.

Testing

9. Summary

- Testing is an essential part of software development.
- Testing ensures that the program works correctly without error and meets the needs of the user.
- A set of test plans need to be produced, usually based on user requirements
- The three types of test data are 'normal', 'borderline' and 'invalid'.
- A black box strategy means only the inputs and outputs are examined
- A white box strategy tests the algorithms and code of the program
- A trace table can be used for white box testing
- Iterative testing means coding, then testing, then coding again until the program is complete
- Final testing is done when the application is considered complete

- Alpha and Beta testing are parts of final testing