Software Testing Issues

Before we take an in-depth look at the process of testing, it is worth perhaps pausing for a moment to reflect upon the implications of failing to adequately test our software systems. As Electrical and Computer Engineers we, perhaps more than computer science students, will probably find that much of our software is developed for Real-time, Embedded and Safety Critical applications.

These in particular represent the most difficult of all software systems to adequately test for reasons that are outlined below. Their failure in the field can and often does have far reaching and devastating effects on human life.

- Real time systems are difficult to test because they respond to *asynchronous* events and data. In fact the combination of inputs and their relationship to each coupled with the need to provide a response within a specified period of time makes it almost impossible or prohibitively expensive to comprehensively test such systems. Here are two examples of such systems that failed due to inadequate testing.

  ".... In 1999 an airbus A320 aeroplane overshot the runway coming in to land in monsoon conditions. When the plane touched down, there was so much water on the runway that the tyres immediately aqua-planned and failed to spin in the normal fashion. This led the computer to believe that the plane was still in the air and refused the pilots request to engage reverse thrust!!!"

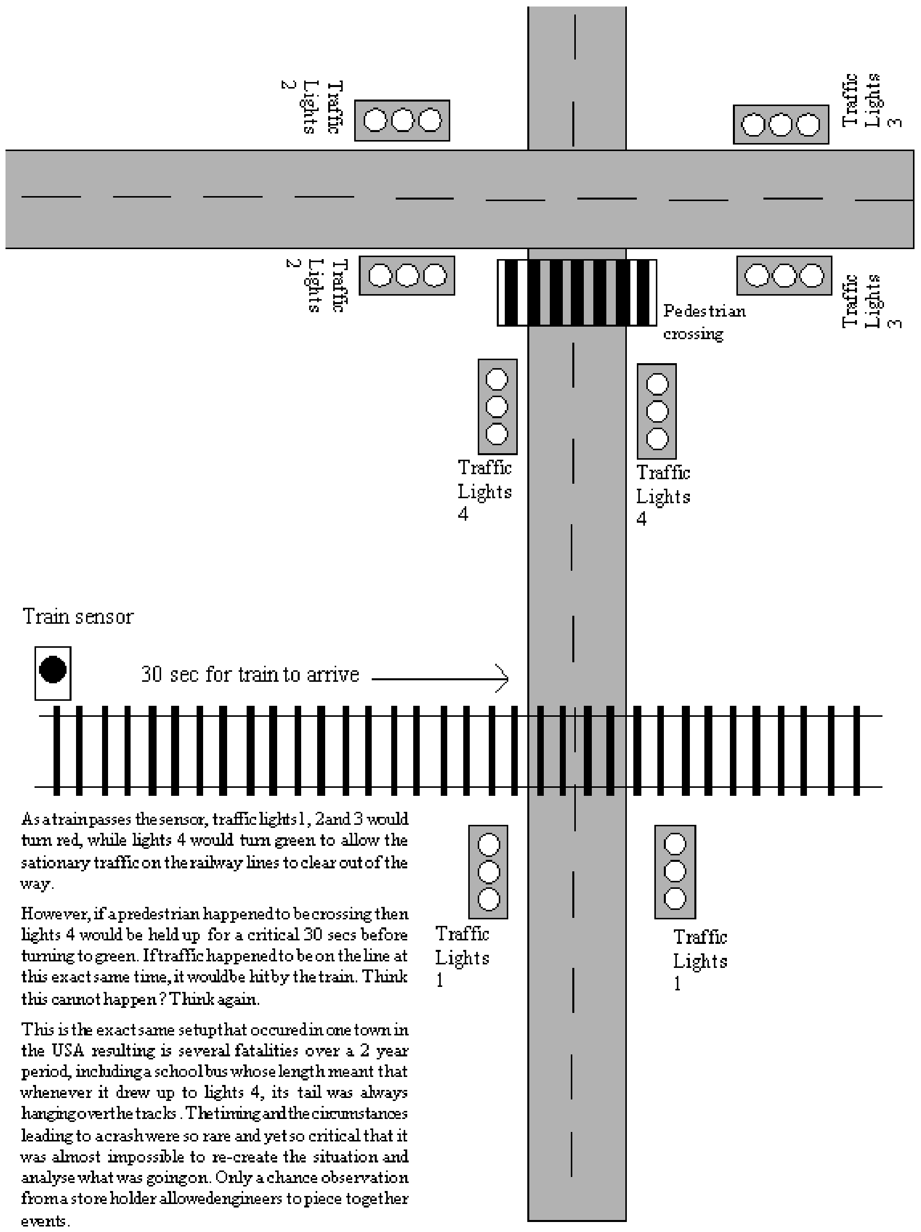
  *"…**Therac-25**: Between 1985 and 1987 at least six people in the US and Canada suffered massive radiation overdoses because of software-related failures in the Therac-25 radiation therapy machine. Three of them are generally accepted to have died as a direct consequence of the massive overdoses of radiation discharged by this machine. Evidence submitted to the investigation suggested that affected patients had received between 13000 and 25000 rads of radiation concentrated at the point of treatment. To put this in context 1000 rads of radiation exposed to the whole body is generally considered sufficient to kill everybody exposed to it. Many other patients suffered horrific and debilitating burns and loss of mobility.*

  An important root cause was a lack of quality assurance, which led to an over-complex, inadequately tested, under-documented system being developed. The design of the system was also severely criticised for removing too many of the physical safety interlocks of its predecessor (Therac-20) and replacing them with software based versions

  A full report of the investigation and findings on this infamous software system failure can be found at http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html)"

- Real-time systems do not lend themselves to the familiar techniques of debugging that we might find useful in other sorts of systems, in particular, you cannot single step or breakpoint the execution of a flight control system for an Boeing 777 when it's 30,000 feet up doing 500 mph !! In fact when a real-time system *does* fail, it can be almost impossible to recreate the exact sequence of events that lead to the failure and thus diagnose the root cause of the problem. In fact there may be nothing left of the system to diagnose!!!

- Many real-time systems are state dependent, meaning that their response to events is based not only upon the event itself, but upon the time dependent history of what has happened before. In other words the same event does not always lead to the same response adding an extra element of complexity to the testing of a system. Here is a good example that caused major loss of life



As a train passes the sensor, traffic lights 1, 2and 3 would turn red, while lights 4 would turn green to allow the sationary traffic on the railway lines to clear out of the way.

However, if a predestrian happened to be crossing then lights 4 would be held up for a critical 30 secs before turning to green. If traffic happened to be on the line at this exact same time, it wouldbe hit by the train. Think this cannot happen ? Think again.

This is the exact same setup that occured in one town in the USA resulting is several fatalities over a 2 year period, including a school bus whose length meant that whenever it drew up to lights 4, its tail was always hanging over the tracks . The timing and the circumstances leading to a crash were so rare and yet so critical that it was almost impossible to re-create the situation and analyse what was goingon. Only a chance observation from a store holder allowedengineers to piece together events.

- Safety Critical systems also have to be *fault tolerant* meaning that they should not just crash or stop working in the event of a hardware failure but should attempt to function in as safe a manner as possible by perhaps offering a reduced performance or functionality. A good example of this is the *limp home* facility built into the sophisticated engine management systems in cars which is designed to keep the car going in the event of a critical sensor failure.

  Safety in critical systems is often dictates the use of *replicated* and *redundant* hardware and software systems. For added safety these are often designed by independent teams working from the same specifications. Thus we probably have a minimum of 3 sets of hardware and software to test and validate.

  This in itself presents difficulties because some hardware systems are quite dangerous to operate without them being adequately tested, e.g. nuclear reactors or fly-by-wire aeroplanes and thus their operation is often tested using simulation techniques. Unfortunately these do not offer the same level of assurance as a real test since simulated faults can be quite different to the real thing.

Verification and Validation Testing

When we attempt to test software, we are attempting to carry out two important activities know as verification and validation of software.

Verification of software refers to a set of activities that ensure that the software correctly implements a specific function, in other words

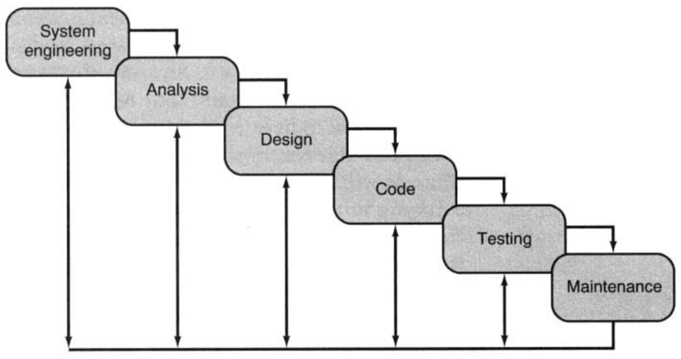
"…Are we building the product right?"

Validation refers to a different set of activities that ensures that the software that has been built to customer requirements. That is

"…Are we building the right product?"

Software Testing

In the classical '*Software Life Cycle*' or '*Waterfall Model'* of Software development, testing is viewed as the last stage of product development prior to the release of a new version of the product to the customer. In practice, testing of the software is almost always carried out concurrently with development, e.g. as code is developed, one or more test cases are written to make sure it works correctly.



During the earlier stages, the emphasis has very much been upon the *creative* processes of analysis, modelling, design and code where the developer "*could show what they could do*" and it follows that he/she (if all has gone well) will feel immense satisfaction and pride in what they have developed.

However, during testing, the emphasis shifts from that of a *creative* process to one of **destruction**. In fact testing is a process **deliberately** intended to *demolish* the software that has been so carefully crafted and honed by the developer and thus represents something of dilemma for the developer.

Quality Assurance and the importance of an Independent Test Group

Whilst it is quite common and acceptable for developers to test the code they have written, it has been shown time and time again that developers invariable tackle the problem by attempting to ***prove their software is correct*** rather than attempting to ruthlessly uncover the bugs that will certainly exist. For this reason, software testing in larger organisations is often performed by an *independent test* or *quality assurance* group.

Testing Objectives

So what are the objectives of Testing? The following points give us some indication of what testing involves.

1. Testing is a process of executing a complete or part program using carefully chosen *test cases* and *test data* with the **deliberate intent of finding an error**.
2. A **successful** test case is one that **uncovers** an as yet undiscovered error rather then one which validates the current designs correctness.
3. A carefully selected test case is one that has a **high probability** of finding an as yet undiscovered error.

In summary,

> *"…The objective of testing is to design sets of test data that systematically uncover different types of errors and do so with a minimum amount of time and effort".*

In other words, testing sets out to **uncover** errors, **not** to prove that the software is correct. During this process it is also important to bear in mind the following rather depressing warning put forward by Dijkstra,

> *"…Testing can only show the **defects** that have been uncovered, it cannot prove that there are no defects".*

In other words, it doesn't matter how much testing you carry out, you still cannot show that software is defect free. It simply means that the tests you have chosen have not exercised the system sufficiently well to *reveal* the defects that will certainly be there.

Furthermore, testing should not be viewed as a safety net for picking up defects or poor quality in analysis or design. In other words

> *"…Testing cannot improve the quality of your design. If it's not there when you start, it won't be there when you're finished. The Therac-25 system proves this"*

How do you know when you have finished testing?

There is no definitive answer to this question, but one response to the above question is:

> *"...You're never done testing. The burden simply shifts from you (the developer) to your customer."*

It's estimated that 30-50% of bugs in a system are uncovered by the customer rather than the developer. Another somewhat cynical, but nonetheless accurate response is:

> *"...You're done testing when you run out of time or money."*

However a more realistic estimate can be obtained by analysing the shape of the defect rate graph as shown below. Here the rate at which defects are uncovered (*inflow*) during testing is plotted alongside the rate at which such bugs are fixed (*outflow*). From this graph we can estimate the likelihood of the software being acceptably free from serious defects. In this case because of the sharp decline in the detection of new bugs we could estimate that a likely release date might be week 8.

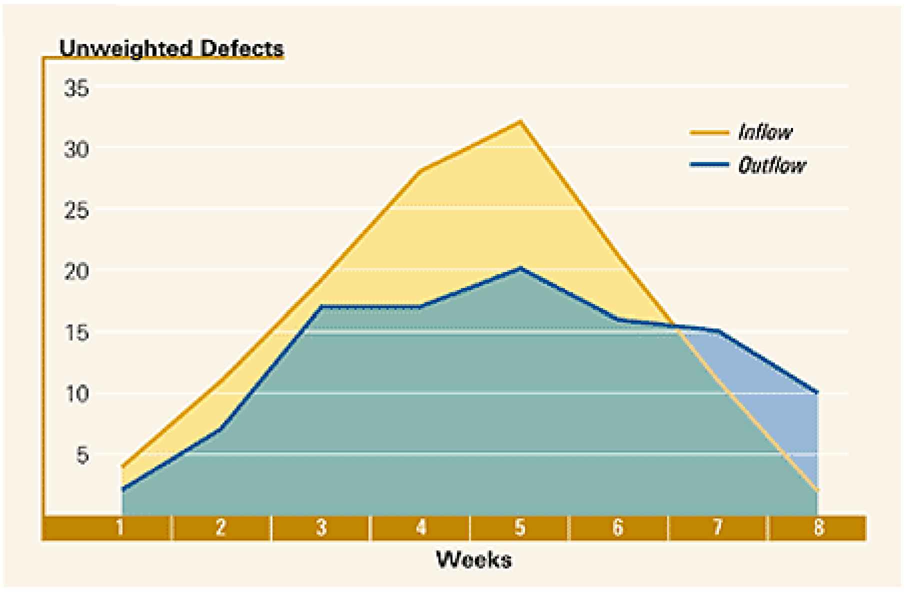
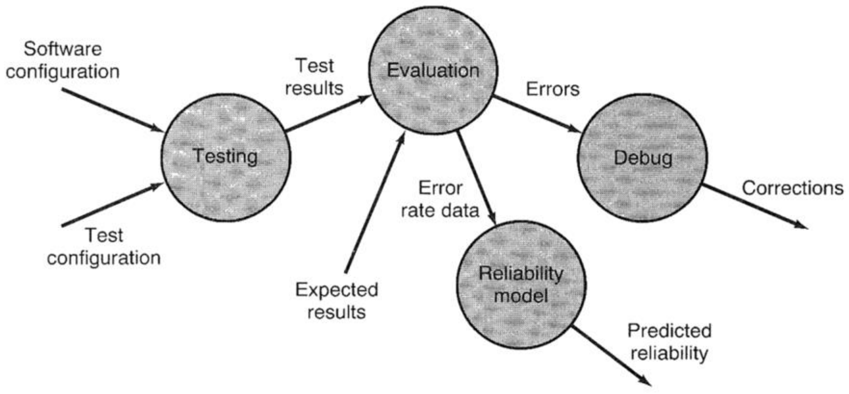


Figure 1: Number of bugs found and fixed each week.

System Testing Process Model

System testing is generally based upon the process model shown below although the actual tests performed may vary considerably depending upon the type of system under test, e.g. on-line, real-time, concurrent, embedded micro-controller etc.



Two types of input feed into the test process:

(1) A *Software configuration* that includes

- The Requirements Specification indicating what the system is supposed to do.
- The Design Specification showing the software architecture of the system.
- The Source code to be tested.

In effect, everything that has been developed, modelled or written.

(2) A *Test configuration* that includes a set of "***test cases***" and "***test case data***" that can be used to exercise the software plus a set of **expected** or **predicted** results/behaviour.

Tests are then conducted and the results are **evaluated**. That is, the actual results obtained from performing the tests are compared with the *anticipated* results or behaviour.

If there is disagreement, then either the results of the test or the anticipated results are incorrect. In the case of the former ***debugging*** can commence in an attempt to correct the cause of the fault in the software.

Reliability estimates for the system can also be predicted such as mean time between failures (MTBF) based on statistical analysis of the failure rate during testing.

So how do we test our system? What techniques can we use and how do these techniques assist us to produce test case data?

Testing Techniques - Black and White Box Testing

As you are probably aware, any engineered product can be tested in one of two ways:

(1) If you know what ***function*** a particular, module or sub-system is supposed to perform, that is you know what task it is supposed to carry out, it should be possible to generate and submit **sample test data** to the module and compare the results it produces with predicted answers.

For example if you were faced with the task of testing a module written to calculate the area of a circle then you could feed in a radius value of 2.0 and check that the module produces the predicted answer 12.56637

This technique is known as '***Black box***' testing because you are in the dark as to how the module actually carries out its requested task, that is, the implementation of the module is hidden from the tester and is not considered.

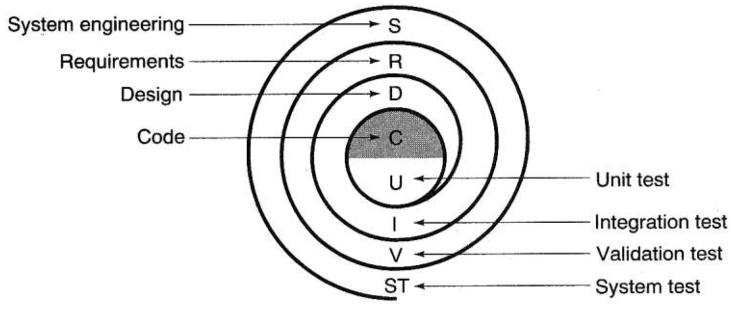
Black box testing thus alludes to tests that are performed at the ***modules interfaces***, i.e. it considers what data goes in, and what results come back or what operations are performed.

(2) However, if you can see, or have access to the internal workings i.e. the source code of the module, perhaps because you have designed or written that module, then you are able to test the correct functioning of all the *statements*, *data structures*, '*if-else*' tests and '*do-while*' loops contained within it. This technique is known as "***White Box***" testing because you can *see* the code under test and can generate tests specially to exercise it.

Combining these two fundamental test techniques it should be possible to verify the correctness of a software system from the smallest component such as a function, to a fully fledged system.

Testing Techniques

Both black and white box tests are employed at various stages of program development and testing. The illustration below highlights that a range of test techniques do in fact exist to verify/validate each stage of the software development process.



- Beginning at the vortex of the spiral we have ***Unit Testing*** which concentrates on each **unit** or **module of code**, typically the **functions** and **subroutines** in your programs and attempts to verify their correctness. Unit testing is performed on individual functions or modules of code and is typically performed immediately after the module/function has been written, i.e. before it is integrated into the rest of the system. Quite often it is the developer that performs these tests

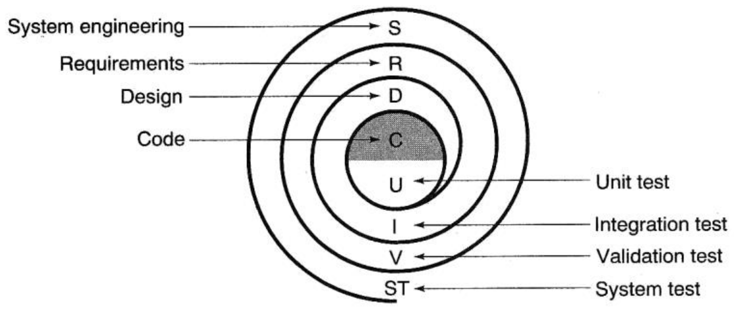
  Unit testing is conducted using both ***black*** and ***white box*** testing techniques. That is, we exercise the internal workings of a module/unit to check that we get the correct answer and that the logic and data structures are not corrupted during its execution.

- Once individual modules have been tested, testing progresses by moving outward along the spiral to ***Integration testing***, where the focus is primarily upon making sure that modules communicate and interact correctly with other modules in the system to build higher-level functionality. Techniques here include

  - **Bottom-up integration.**
  - **Top-down integration.**

  Integration testing is predominantly a *Black Box* test technique exercising the interfaces of the modules and sub-systems. Such test highlight any inconsistencies between the co-operating modules.

  For example, take a Pentium IV processor. Hardware Unit testing will show it works, so does the motherboard, but integration testing will show up if the processor and motherboard are compatible.



- Taking another turn of the spiral, we encounter ***validation testing***, where the **requirements** of the system are validated against the software that has been constructed. That is, we answer the question

  "…Have we built the product that was customer wants?"

- Finally, we arrive at ***system testing***, where the software and other system elements are tested as a whole. This is sometimes known as ***acceptance*** testing. Techniques here include amongst others, ***Alpha*** and ***Beta*** testing.

  Here we can perform test that are only possible once the software is installed at the user premises and they are using it e.g. Graphical user-interfaces, forms, dialog boxes etc, do they ask the right questions, can the user escape from them or correct their entries, stress/load testing (can it cope with 1000 transactions per hour etc), does it interact with the customers database and other computers etc.

- Once software has been released we enter the maintenance phase where new or modified functionality may be added to the system over time. Here *regression* testing is used to ensure that the software does not regress or fall back into a worse state than that which existed before the modifications were carried out.

Unit Testing – A Practical Guide

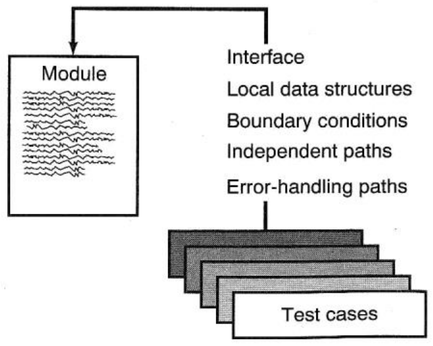
Unit testing focuses effort on the smallest unit of software design, the module or function. Using the detailed design description as a guide, important control paths are tested to uncover errors within the boundary of the module itself. Testing of various units/modules can be conducted in parallel.

Unit Test Considerations

A list of possible Unit tests that could be performed on a module is given below.

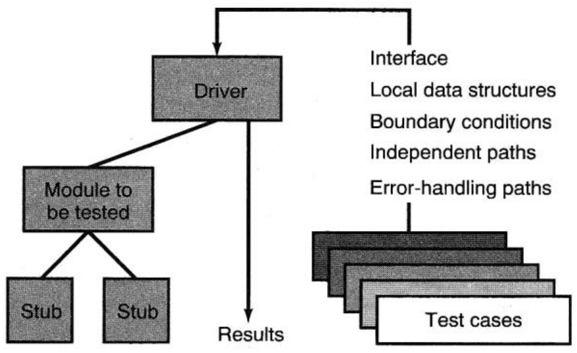
1. Test the modules *interface* to verify that data flows correctly into and out of a module, i.e. call the module, pass it data and check the result produced.
2. Examine the *local data structures*, such as arrays to establish that they are not violated (e.g. *array bounds exceeded*) during the execution of a module.
3. Exercise all *independent paths* (i.e. % code coverage - https://www.guru99.com/code-coverage.html) through the control structure to ensure that all statements in a module have been executed at least once (*Basis Path Testing*).
4. Exercise the systems response to data that is at the *extremes* of its range of validity (*Boundary Value Analysis*).
5. Test all *error-handling paths*, that is, check that the system deals correctly with any erroneous data or conditions that arise during execution.

It may well be the case that you will have to generate independent/separate categories of test cases to perform each of the five unit tests described above. The diagram below illustrates this.



Unit Testing – Stub and Driver Functions

Because a unit or module is not a stand-alone program, a **driver** and/or **stub** module must be developed for each module under test. Again this is illustrated below.



In most applications a *driver* is nothing more than a primitive program or function that accepts test case data from the tester and passes such data to the module under test. In addition, it then has to transport the relevant results produced by the tested module back for inspection by the tester.

Stub functions replace modules that are *subordinate*, i.e. those modules that are *called* by the module under test. A *stub* function must use the subordinate module's interface and may have to carry out minimal data manipulation, print verification of entry and return dummy results to assist in the testing of the main module.

Note that both *drivers* and *stubs* represent an overhead, i.e. both represent software that must be written to carry out testing but which will not be delivered with the final software product. If drivers and stubs are kept simple, actual overhead can be kept relatively low.

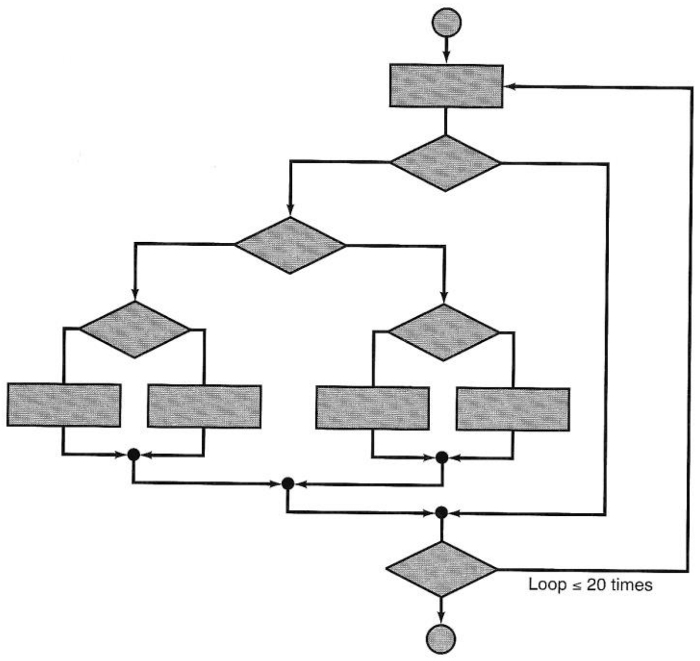
Unfortunately, many modules cannot be adequately *unit-te*sted with simple overhead software perhaps because the interface or number of sub-ordinate modules is complex or large.

In such cases, complete unit testing may have to be postponed until the integration test step, where drivers or stubs are replaced by the real modules.

Unit Testing Techniques - White Box Testing

At first glance it would seem that very thorough white box testing would lead to 100 percent correct programs. After all, we only need to define all possible paths of execution within the module/program, develop test cases to exercise them, and evaluate the results, to completely exercise the program logic and demonstrate its correctness.

Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. For example, consider the very simple flow chart shown below.



Such a flow chart could be implemented with perhaps just 50 lines or less of 'C' code with a single '*do-while*' loop and just 4 '*if-else*' tests. However, a quick calculation reveals that there are 95,367,431,640,625 or 5 (the number of individual routes through the program), raised to the power 20 (the number of iterations of the '*do-while*' loop) possible paths that could be executed!

To put this number in perspective, if we assumed that an *automated test processor program* existed that could analyse a program and develop a new test case, execute it, and evaluate the results within 1 millisecond, working 24 hours a day, 365 days a year, the processor would work for over 3000 years to test the relatively simple program represented above. Exhaustive testing then is **just not an option** for large software systems.

White box testing thus concentrates on generating a *limited number* of *carefully selected test cases* to exercise important execution paths and decisions within the program. This is often all that will be required to give the developer ***a high degree of confidence*** that the module is working correctly, especially when these tests cases are combined with black box and other testing techniques.

Practical White Box Testing

Practical white box testing techniques involve the creation of test data that will

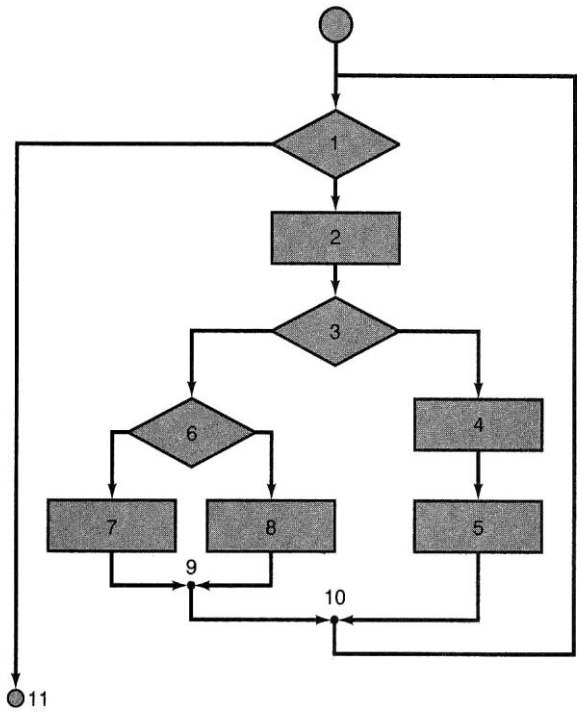
1. Guarantee that all statements within a module have been exercised *at least once*.
2. Guarantee that all '*if-else*' tests and '*do-while*' loops have their *true/false* results evaluated at least once.
3. Execute all loops at and within their *boundaries* i.e. their start and end values and within their operational bounds.

More Practical White Box Testing - Basis Path Testing

Basis path testing is a white box testing technique that enables the tester to derive a *logical complexity measure* for a module and use this measure as a guide for generating a set of fundamental test cases (known as the **basis set**).

Such test cases are guaranteed to execute every statement in the program **at least once** during testing. The techniques can be used to assess for testing, a module design that exists either as a flowchart, pseudo code or program source code.

As an example of Basis Path Testing, consider the following flow chart. Here, each decision box (shown as a diamond) represents a simple *logical test* such as an '*if*' test, or a '*do…while*' loop within the program. Rectangles represent sequence. Each box is given a (node) number, not forgetting to number the end of the module (11 in this case). Where two sequences meet, an *implicit node* can be introduced and numbered (see nodes 9 and 10).



If you were to follow this flow chart, you would, using intuition and common sense, realise that there are 4 individual routes of execution through this module. These routes involve traversing the following nodes.

Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11

In fact, it can be shown that the number of independent paths of execution in a module flowchart is equal to the (**number of decision diamonds +** 1) in other words it is equal to the number of (*if…else* tests + *do…while* loops + 1) in the modules code. This metric was first proposed by Thomas McCabe in 1976 and is also known as the cyclomatic complexity number.

This numeric value is both simple to derive and yet very important since it tells us in this instance that we need to generate a minimum of 4 test cases to exercise the module in accordance to the rules of basis path testing. (*It also gives an indication of where you should be spending most of testing time – on the more complex modules*)

Any program or Pseudocode listing can also be analysed in the same way as a flow chart to give a measure of its cyclomatic complexity and hence an indication of the number of unique execution paths within the module. For example, consider the 'C' code segment shown below. Note that in the code, statements have been numbered

```
while (a != 0) {              1    (decision 1)
    a = a –1                  2    (sequence)
    if ( b < (a + c)) {       3    (decision 2)
        c = c + 1             4    (primary sequence)
        print b               5    (sequence)
        print c               6    (sequence)
    }                         7    (sequence)
    else {                    8    (alternative sequence)
        if( d > x ) {         9    (nested Primary decision 3)
            print x           10   (sequence)
        }                     11   (sequence)
        else {                12   (nested Alternative decision)
            print x*x         13   (sequence)
            print  x*x*x      14   (sequence)
        }                     15   (sequence)
        statement             16   (sequence)
    }                         17   (sequence)
}                             18   (sequence)
                              19   (end of module)
                              20
```

By counting the number of individual decisions made within the program (3) we arrive at a complexity measure of **4** for the module, that is, there are four individual routes of execution within the program, these are.

Path	**Route**	**Description**
1:	**1-19**	**(straight in and out again)**
2:	**1-2-3-4-5-6-7-18-1-19**	**(1st if test true)**
3:	**1-2-3-8-9-10-11-16-17-18-1-19**	**(1st if test false, 2nd if test true)**
4:	**1-2-3-8-9-12-13-14-15-16-17-18-1-19**	**(1st if test false, 2nd if test false)**

To exercise the above paths through the module, we can simply set up the pre-requisite test case data as shown below.

Path	**Route**	**Pre-requisites or Conditions**
1:	**1-19**	ensure that 'a' is set to 0
2:	**1-2-3-4-5-6-7-18-1-19**	ensure that 'a' is set to value 1 (i.e. a non zero value) and that b < (a + c)
3:	**1-2-3-8-9-10-11-16-17-18-1-19**	ensure that 'a' is set to value 1 (i.e. a non zero value) and b >= (a + c) and d > x
4:	**1-2-3-8-9-12-13-14-15-16-17-18-1-19**	ensure that 'a' is set to value 1 (i.e. a non zero value) and b >= (a + c) and d <= x

Other Unit Test Techniques - Testing Module Interfaces

When testing the modules interface, consider the following check list for guidance. In the following discussion, a modules *parameter* refers to how the module declares the data that it expects to be given when it is called. For example, in 'C' a module which calculates the volume of a circular based cone might be written as shown below, where the two parameters '*radius*' and '*height*' *are* declared as *floats*.

```
float volume( float radius, float height)
{
    . . .
}
```

The *arguments* to a function refer to the actual data that is passed into the function when the function is called. For example, the following statement calls the function 'volume()' above with the arguments 2.0 and 5.0.

```
x = volume( 2.0, 5.0 ) ;
```

The following tests could be conducted on this modules interface

- Do the *numbers of parameters* declared in the function itself equal the *number of arguments* passed into it when it is called from a higher-level module? Any half decent compiler would be able to flag this violation as an error at compile time.

- Do *the types of the arguments* match the corresponding *types of the parameters* or can they be automatically converted by the compiler without loss of accuracy of precision. For example, '2.0' above represents a '*double'* in 'C', while the function 'volume()' expects a '*float'*. This is Ok as a 'C' compiler will convert a *double* to a *float* but could the resulting loss of value and/or precision affect the result? A good compiler would generate a warning for this

- Do parameter and argument *units* match? For example, is the function expecting say, a velocity value measured in feet/sec or metres/sec and does the calling function know which? In other words check for a miss-match. In 1999, the **Mars Orbitor**, a joint NASA/European Space Agency project was said to have crashed onto the surface of mars (instead of orbiting it) because NASA used imperial units or measurement while ESA used metric. Remember, a function just gets given a number, it doesn't know if it represent Inches, or Millimeters, but you will notice when it crashes !!

- Do *the order of arguments* passed to a function *match the order of parameters* declared in that function? For example, when calculating the volume of the cylinder has the function been written to expect the height to be the first argument followed by the radius, or vice-versa. Does the calling program know?

Other Unit Test Techniques - Testing Modules External Interfaces

When a module performs external I/O, additional interface tests must be conducted.

- Where file access is performed, check that the file attributes are correct? For example are the disk/path/file names compatible and correct for the particular operating system and are permissions specified to access the file in the required manner correct? For example read/write/append.

- Are the files opened and closed in the correct sequence and are all opened files closed at the end?

- Does the format specification match the I/O statement? For example, is the program using the correct printf(), scanf() format string to write/read the data to/from a file in the 'C' language?

- If the program reads 'records' from the file, are they read in the correct sized chunks? In other words, make sure your program is not reading ¾ of a record, as this will mean that the next record read will be a mixture of two records and therefore rubbish.

- Does the program correctly deal with errors that might occur when reading from or writing to a file?

- Is the program written to correctly detect the End-of-File (EOF) condition, i.e. can it detect when there is no more data to be read?

Other Unit Test Techniques - Testing Local and Global Data Structures

The local and global data structures for a module are a common source of errors. Test cases should be designed to uncover errors in the following categories:

- Check that all variables are initialised before they are used. In 'C' an un-initialised local variable assumes a *random* value. Most good compilers will be able to flag this as an error.

- Check the spelling of variable names to make sure that they access the correct variable. In B.A.S.I.C, if you miss-spell a variable name, then a new or different one might be created or used.

  This particular category of error is particularly difficult to track down. In 'C' for example failing to declare a local variable '*result'* might mean that a global variable of the same name is used instead.

- Make sure that the variables you introduce to hold the results of calculations are correct for the type of data that will be assigned to then. For example introducing an '*int*' to hold the result of a floating point calculation is asking for trouble.

- Underflow, overflow, rounding and addressing exceptions.

  i. Underflow errors occur when the result of a calculation is too small to be represented by a particular data type and it may be truncated to 0. e.g. assigning an Integer the value 0.25 or a float the value 1.0e-128
  ii. Overflow errors occur when the value is too large. For example assigning a 16 bit integer the value 2321231231 or a float the value 1.23e+300
  iii. Rounding errors occur as a result of a certain data type inability to exactly represent a given value, for example, assigning a float the value 1.12497 may actually result in it being assigned the value 1.125 etc.
  iv. Addressing errors occur when a program violates its own memory (array bounds exceeded or inappropriate used of pointers in C/C++)

Among the more common errors in computation are

- Misunderstood or incorrect arithmetic precedence, e.g. assuming one operator will be evaluated before another when in fact the opposite is true (use '()' to force the order in an unambiguous manner).

- Mixed-mode arithmetic. That is mixing for example, *int's, floats* and *chars* in the same mathematical expression. This can lead to incorrect truncation, overflow, underflow and rounding errors.

- Incorrect symbolic representation of an expression. For example using the wrong operator such as using '=' in 'C' to mean equal to instead of '= ='

Other Unit Test Techniques - Testing Error Detection Paths

There is a tendency to incorporate error handling into software and then never test it. Among the potential errors that should be tested when error handling is evaluated are

- Is the error description intelligible and does it provide sufficient information to assist in the location of the cause of the error? A simple statement like "Error: Core dumped !!!" typical of the sort produced by early Unix implementations is not very helpful.

- Do the error messages refer to the true error or something different?

- Does an error condition cause the operating system to intervene prior to detection? For example, does your array access get validated by your software, or is it left to the Memory Management Unit to detect and lead to the shutdown of your program?

- Does your error detection code deal with the error correctly? For example, if a user is asked to enter a number in the range 0-5, what does it do in response to the number 6, Crash and Burn?

Other Unit Test Techniques - Loop Testing

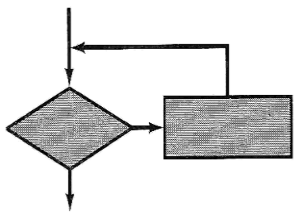
Loop testing is another white box testing technique that focuses exclusively on the validity and testing of loop constructs. Four different classes of loops can be defined for the purpose of testing:

- Simple loops,
- Concatenated loops,
- Nested loops
- Unstructured loops

Each of these types of loops can be tested in the following manner.

Unit Testing - Simple Loops.

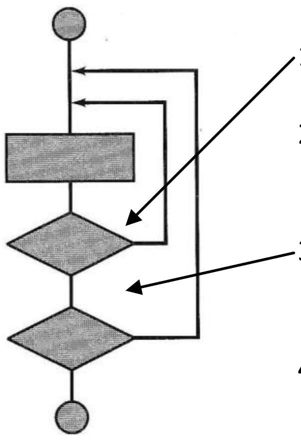
The following set of tests should be applied to simple loops, where 'n' is the maximum number of iterations of the loop



1. Skip the loop entirely.
2. Execute 'm' passes through the loop where m < n.
3. Execute 'n - 1', 'n' and attempt to get the program to perform 'n + 1' iterations of the loop (i.e. can the loop check its termination status correctly)

Unit Testing - Nested Loops

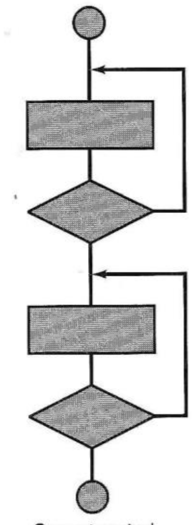
If we extend the test approach for simple loops to nested loops, the number of possible tests grows geometrically as the level of nesting increases. This would result in an impractical number of tests. It has been suggested the following approach will help to reduce the number of tests:



Nested loops

1. Start at the innermost loop and set all outermost loops to their minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values.
3. Work outwards, conducting tests for each outer loop, but keeping all other outer loops at their minimum values and other nested inner loops at their "typical" values.
4. Continue until all the loops have been tested.

Unit Testing - Concatenated Loops


Concatenated loops

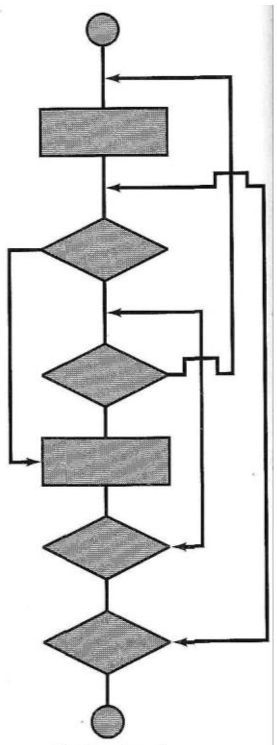
These loops can be tested using the approach defined for simple loops above, if each of the loops is independent of the others.

However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.

When the loops are **not** independent, the approach applied to nested loops is recommended.

Unit Testing - Unstructured Loops

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs. That is, remove the *goto's* and replace with appropriate *if-else* tests and *do…while* loops.


Unstructured loops

Exercise.

The following rules are used by UK Universities to determine a students degree classification based upon the aggregate/mean of their set of final year grades.

An aggregate of >= 70% is awarded a 1st class degree.
An aggregate of >=60 and <70 is awarded a 2:1 degree.
An aggregate of >=50 and <60 is awarded a 2:2 degree.
An aggregate of >=40 and <50 is awarded a 3rd class degree.
An aggregate of >=35 and <40 is awarded a pass degree.
An aggregate of < 35 indicates the student has failed.

A simple *C++* code fragment to perform the task of awarding the classification is given below

```
void  CalculateClassification(double Aggregate)
{
    string result;

    if(Aggregate >= 70)
        result = “1st” ;
    else if(Aggregate >=60)
        result = “2:1” ;
    else if(Aggregate >=50)
        result = “2:2” ;
    else if(Aggregate >=40)
        result = “3rd” ;
    else if(Aggregate >=35)
        result = “Pass” ;
    else
        result = “Fail” ;

    return result;
}
```

- Using Basis Path testing techniques analyse this module and determine the number of independent test cases that will be required to exercise each statement within the module.
- For each path, determine what test data that will be required to force execution of the module along that path
- For each test set, how will you know whether the module has worked correctly or not? That is, what results would you expect to see after each test to verify whether or not the module is functioning correctly?
- Now think about this. Suppose that the above program contained a bug and the 1st statement was incorrectly written as **if(Aggregate > 70)**. Would your tests be able to pick up this bug? What would happen if a student had an aggregate mark of exactly

70%? What conclusion can you draw from this? What additional tests could you incorporate into your testing to detect this class of problem?

Integration Testing

Having performed unit tests of the software modules, we now have to test them together. It's a legitimate question to ask why there should be a problem. After all, if the modules work correctly, surely they should all work when we put them together?

Unfortunately, this is rarely the case. The problem is of course, in the "*putting them together*" or interfacing of the modules. This is a bit like the engine division and the gearbox division of Ford Motor Company, testing their engines and gearboxes in isolation and then expecting, as if by a miracle, that they will work together when they have never even seen each other. Some examples of what can go wrong when software is integrated are given below.

1. Data can be lost or corrupted across an interface i.e. when data is passed from a *calling* module to a *called* module, it may get corrupted due to rounding, truncation etc errors or simply that one module changes it before passing it to another. In addition, the units for the data may be inconsistent pounds vs kilograms etc. and the order of data passage may also be incorrect, e.g. data relating to the height and radius of a circular cone may be passed in the wrong order.

2. One module can have an inadvertent, adverse affect on another, e.g. they could both share global variables for different purposed. Such problems will go undetected during unit testing and only shows up when both modules execute in the same program

3. Sub-functions, when combined, may not produce the desired major function.

The list goes on and on.

Integration Testing Techniques and Approaches

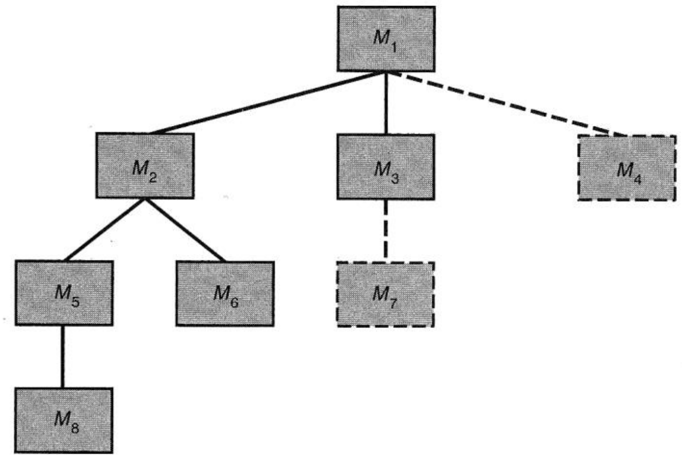
The objective of integration testing is to take unit-tested modules and build them into a complete program structure that can be tested as one. There is often a tendency towards "*big bang*" integration, where all modules are combined in advance and then tested as a whole.

The only attraction of this approach is that it eliminates the need to write stub and driver functions to assist with testing. However, the result is usually chaos, since when an error occurs, correction is difficult because the isolation of the cause is complicated by the vast expanse of the entire program.

Incremental integration involves constructing and testing the program in small segments where errors are easier to isolate and correct; interfaces are more likely to be tested completely, and a systematic test approach may be applied. In the sections that follow, a number of different incremental integration strategies are discussed.

Top-Down Incremental Integration

With Top-down incremental integration, modules are integrated by moving *downward* through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure in either a *depth-first* or *breadth-first* manner as shown below



- The *Depth-first* approach integrates all modules on a major control path into the structure. For example, selecting the left-hand path, modules M1, M2, M5 would be integrated first.

  Next, M8 or if necessary for the proper functioning of M2, module M6 would be integrated. Then the central modules M3 and M7 and right-hand module M4 control paths are built.

- The *Breadth-first* approach integrates all modules directly subordinate at each level, moving across the structure horizontally. From the illustration, modules M2, M3 and M4 would be integrated first. The next control level, M5, M6, M7 etc. then M8 would be built.

The integration process itself is performed as a series of four steps:

➢ The main control module is used as a test *driver* and *stubs* are substituted for all modules directly subordinate to the main control module.

➢ Depending on the integration approach selected (i.e., *depth* or *breadth* first), subordinate *stubs* are replaced one at a time with actual modules.

➢ Tests are conducted as each module is integrated.

- On the completion of each set of tests, another *stub* is replaced with the real module.

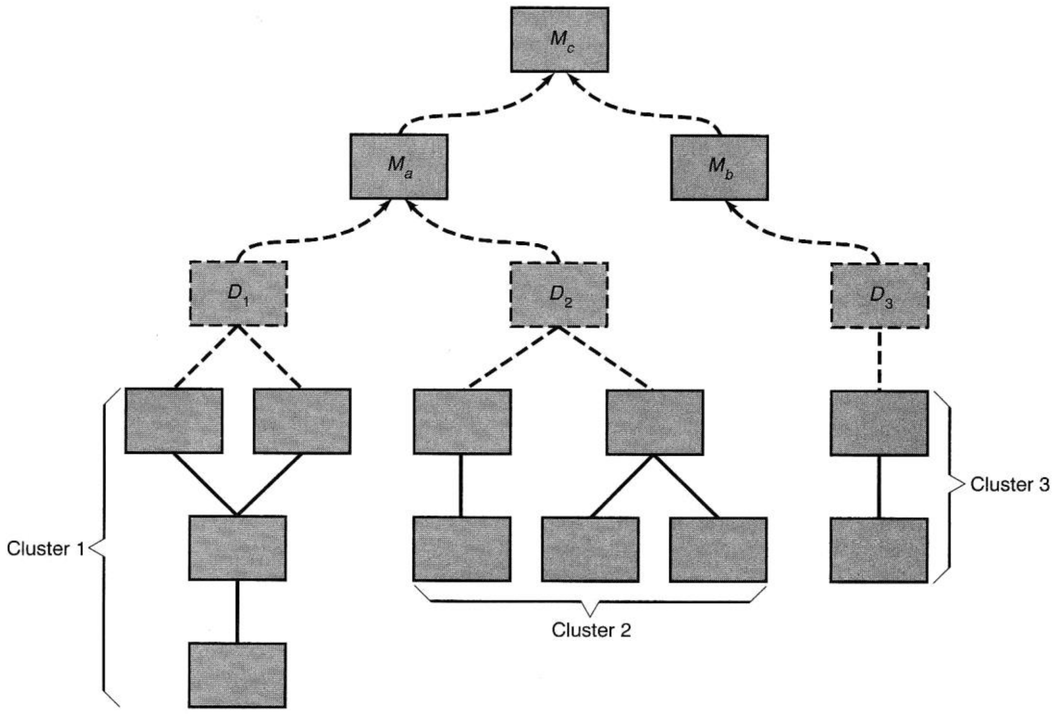
The process continues from step 2 until the entire program structure is built.

Bottom up Incremental Integration

Bottom-up incremental integration testing begins construction and testing with the lowest level modules in the control structure. Because modules are integrated from the bottom up, processing required for modules subordinate to a given level is always available and the need for *stub functions* is eliminated. A bottom-up integration strategy may be implemented with the following steps:

- Low-level modules are combined into *clusters* that perform a specific software sub-function.
- A *driver* module is written to co-ordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated below



Modules are combined to form clusters each of which is tested using a driver *Dn* (shown as a dashed block). Modules in clusters 1 and 2 are subordinate to *Ma* therefore drivers *D1* and *D2* are removed and the clusters are interfaced directly to *Ma*. Similarly, driver *D3* for cluster 3 is removed prior to integration with module *Mb*. Both *Ma* and *Mb* will ultimately be integrated with module *Mc*,

Integration Test Techniques - Black Box Testing

Black box testing methods focus on the functional *requirements* of the software, that is, they check that the software *does what it is supposed to do*?

Black box testing is *not* an alternative to white box techniques. Rather, it is a complementary approach that is likely to uncover a completely different class of errors than those likely to be detected with white box testing methods.

Black box testing attempts to uncover errors in the following categories:

- Incorrectly operating functions, that is, ones that do not match their requirements i.e. they do not do what they should.
- Interface errors, that is, any mismatching or incompatibility of data passed between one function and another.
- Initialisation and termination errors, that is, functions that do not start up and terminate with the correct data/results.

Practical Black Box Testing Techniques – Equivalence Partitioning

Equivalence Partitioning is a very powerful black box test technique. It is based upon the idea that it should only be necessary to subject a module to a carefully chosen combination or *sub-set* of valid input test data in order to have a high degree of confidence that the module will work correctly with a much broader range of valid inputs. In other words it suggests that we don't have to write an infinite number of test cases to test the systems response.

The concept is best explained by an example. Suppose we were faced with the task of testing an elevator responsible for **80 floors**. Inside the elevator are 80 buttons. Equivalence Partitioning suggest that we would not have to verify the systems correct response to each and every press of these 80 separate floor buttons, since it is **probably** the case that if the elevator responds correctly to say floor button **15** or **66**, then there is a high degree of probability that it will also respond correctly to all other floor buttons.

This degree of confidence stems from the fact as programmers we tend to create *generic algorithms* as solutions to specific tasks. In other words a programmer will tend, more often than not, to come up with ***one fragment of code*** for dispatching an elevator to *any* of the 80 floors rather then generating 80 separate, independent code fragments to deal with each floor button.

This assumption (if proven correct) would allow us to significantly reduce the number of tests we have to perform to check the response of the system to various input data. In other words, if the code works correctly for a single item or *sub-set* of its valid input data then it will probably work correctly all of them. Thus testing the elevators response to floor buttons such as 15 or 66 is going to be **equivalent to** testing the elevators response to all floors buttons.

Equivalence Partitioning - Classifying Input Data

Before we can make use equivalence partitioning, it is important to classify the nature of data processed by a module so that we can generate suitable equivalence test cases. Once we understand what the data is, we can generate sub-sets of the data as being equivalent to all others.

Each and every item of data we **input** to a **function**, whether it comes from the keyboard, file, or in the form of arguments passed to it during a call, can be **classified** on the following basis:-

- The *value* has a *specific length*. For example, a 16 digit bank account number, a 10 digit telephone number or say a 20 character City Name.
- The *value* has either an *infinite* range or is limited to a *bounded range*. For example the values 1-80 might represent a bounded range of elevator floors
- The *value* is one of a *finite set* of related values, e.g. the set of colours {Red, Green, Blue}, {True/False}, {Yes/No}, {Present/Absent}.
- The *value* is Optional. For example, a Title, a Middle Name, a Zip code

Give this classification, how do we design suitable equivalence test to exercise our system with a high degree of certainty?

Guidelines for Generating *Equivalence* Test Cases

1. If the item of test data is such that it presents a *specific length value*, then generate test data with **one valid** and **two invalid length values**. For example, a function designed to accept a 10 digit telephone number, might be tested with a 9, 10 and 11 digit numbers to see how it responds.

2. If the item of test data is such that its value always lies within a specified *bounded range*, then test the system response to one valid and two invalid values. For example, the code used to dispatch an elevator to a floor could be tested with the floor numbers -1 (illegal), 5 (legal) and 81 (illegal).

3. If the item of test data is such that it is a member of a set, then provided the set is small, test **all** valid set members and one **invalid** one.

   For example assuming the existence of a set of colours comprising {Red, Green, Blue}, one might generate a test which introduced the program code to the items Red, Green, Blue (all legal) and Yellow (illegal). However if the set is large, just choose one valid and one invalid member from the set.

4. If the item of test data is optional then test the software response to the **inclusion** or **omission** of the data.

Practical Black Box Testing Techniques - Boundary Value Analysis

Experiments have shown, that a greater number of functional errors tend to occur at the *boundaries* or *limits of* a functions input data. I am sure that we have all observed this phenomenon in practice, when we write programs that iterate around a loop one too many or one too few times, or we write a '>' conditional test in our code when we meant '>=', or perhaps stray just beyond the limits of an array boundary.

For this reason, boundary value analysis (BVA) has been developed as a testing technique to exercise the function's responses to data at the extremes of its validity.

Guidelines for Generating *BVA* Test Cases

1. If an input specifies a *range of permissible values* bounded by the values '*a*' and '*b*', test cases should be designed with values just above and just below '*a*' and '*b*', respectively. For instance if the range of acceptable floor numbers in an elevator dispatcher is 0-11 then floor numbers of (–1, 0 and 1), and (10, 11 and 12) should be used.

2. If an input specifies a discrete set of values, e.g. 2, 5, 7 and 9, then test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested. E.g. (1, 2 and 3) and (8, 9 and 10) should be used.

3. Guidelines 1 and 2 should also be applied to exercise the extreme limits of the output or responses of a system. For example, assume that a process control system is monitoring temperature and is designed to sound an alarm if the temperature falls outside a particular range. Tests should be conducted to ensure that the system sounds the alarm when these conditions occur.

4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundaries and make sure the software copes with illegal values.

Most software engineers intuitively perform BVA to some degree. By applying the guidelines noted above, BVA would be more complete, thereby having a higher likelihood for error detection.

Exercise: Imagine an automated banking application where the user can "dial" the bank using his or her personal computer, provide a six-digit password, and then, using a series of keyword commands, trigger various banking functions such as check balance, print statement, order check book etc. The software supplied for the banking application accepts data in the following form:

- **A Telephone number comprising of**
  - **Area code** – Either absent or a three-digit number.
  - **Prefix** – A three-digit number not beginning with 0 or 1.
  - **Suffix** – A four-digit number.
- **A Password** - A six-digit alphanumeric value.
- **A Command -** "Balance," "Statement," "Order Check Book" etc.

To test the response of the system, the following tests will be generated.

Telephone Number Test - Area Code

- The Area Code is optional, that is, it is either present or it is not, thus the system would be tested for both.
- Likewise if the area code is present, it must be three digits. This is an example of an item of *specific length data*, so a test would be made with an area code of length 2, 3 and 4 digits to check how the system responds.

Telephone Number Test - Prefix:

- This item of data specifies that the prefix must be 3 digits in length and, because it cannot commence with a 0 or a 1, any value in the range 200-999 will suffice. Therefore test the software response to a prefix with the following example values 20 (illegal length), 199 (just outside boundary of acceptable input), 500 (legal length/value equivalence test), 1000 (illegal length and value just outside boundary).
- For completeness the digits 0xx and 1xx represent a small set of illegal inputs so the systems response to prefixes beginning with 0 and 1 could be tested (the latter will be have been tested by the test conducted above)

Telephone Number Test - Suffix:

- This is required to have a length of 4 digits, so test the software response to 3, 4 and 5 digit suffixes

Password:

- This is required to have a length of 6 characters so passwords of length 5, 6 and 7 would be tested.

Commands:

- These form a small subset, so test all three valid commands and one invalid one.

White Box Testing Techniques - Assertions

Assertions represent a useful tool in the armour of the tester and developer alike. An assertion is *any expression that must evaluate to true*, and are frequently employed by developers to validate the data passed to and returned by modules.

For example, a simple assertion for a 12 storey elevator control system is that there can never be a request to travel to a floor less than 1 or greater than 12. A violation of this rule or assertion would almost certainly have catastrophic results for the elevator, never mind anybody who happened to be travelling in it at the time!

When the programmer writes the control software to move the elevator between floors, he or she would build such an assertion into the 'transport' program module that is responsible for moving the elevator to a new floor.

If the elevator 'transport module' were asked to move the elevator to an illegal floor, such as 26, the assertion would be invalid and the module could flag this as an error and terminate the program. For example, consider the transport module shown below, which includes an assertion statement to check the incoming floor request and also to check that the elevator is still within range after it has finished!

```
void   transport( int Request)
{
    ASSERT(Request >=1 && Request <=12)

    /* Move to requested floor */

    ASSERT( current_floor >= 1 && current_floor <= 12)
}
```

Now, if the '*system controller'* module were asked to transport the elevator to an illegal destination, as demonstrated below

```
void system_controller()
{
    …
    transport( 23) ;       /* illegal floor number */
    …
}
```

The 1st assertion in 'transport()' will fail and the program will be aborted with an appropriate indication of the assertion that has failed. Likewise, if the 'transport' module contained a bug that meant that it attempted to travel to an illegal floor number, this violation would be detected by the second assertion.

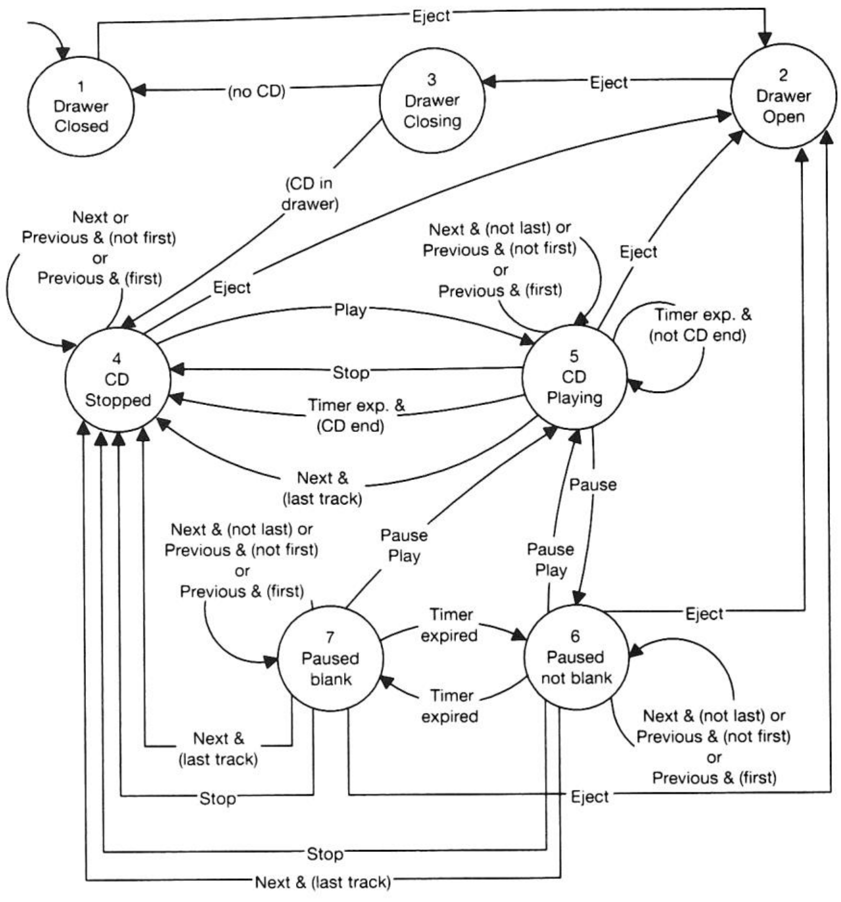
An important and useful feature of assertions is that they can be turned on and off with a simple compiler switch/flag and thus easily enabled for testing.

State Machine Testing

Testing state machines represents an interesting challenge for the software engineer. Take the state chart for the CD player shown below as an example. Here, various events can trigger the CD player from one state to another, some events are externally applied, for example, the user pressing the play button, while others, such as the 'timer expired' as generated internally.

The problem is that many of these events are triggered asynchronously and can occur at any point in time. Suggested techniques for testing a state machine include

1. Exercising the system to ensure that all paths or changes of state occur correctly.
2. That the appropriate actions that should take place, either when migrating to a new state, or occur while *in* that state, do in fact take place.
3. That all events are recognised by those states designed to act upon them, and ignored by those states that should not act upon them.



Automated Testing

Obviously the above pages describe a wide range of practical test techniques that can be applied to test anything from small simple functions to large complex systems. In large systems, testing software *manually* is prohibitive as the software may have many thousands of functions and there isn't the time or resources to developers to run the tests by hand and examine the results by eye.

Even if one could perform this task once (*which could take days/weeks*), any change to the software means that the tests should all be conducted again (*to make sure we didn't break anything*). This is where automated software testing suites are useful. Here, you write tests in such a way that they yield ONLY pass/fail conditions. We are not interested in values produced by the functions, only that they pass or fail a test.

Many thousands of test cases can be constructed (*often by the developer*) and they can be run at the press of a button. Many large organisations run tests continuously so that they pick up immediately if the code base "breaks".

These two links show how to set up automated testing in Visual Studio for the C++ language. We will use this for our second assignment:-

Good intro: https://devblogs.microsoft.com/cppblog/cpp-testing-in-visual-studio/

More detailed:
https://docs.microsoft.com/en-gb/visualstudio/test/unit-testing-existing-cpp-applications-with-test-explorer?view=vs-2015#objectRef

Two example Visual C++ projects have been created on Canvas using the above techniques. One to test a couple of simple functions, the other to test a Bulb Class similar to the one we development in Lab 0.

Just a heads up that the **linker** option for the test projects both contain an absolute pathlist that needs to be changed according to where your project is located on the C: drive.