Software Testing and Validation

Juha Taina
University of Helsinki
Department of Computer Science

1. Introduction

- The software engineering approach, as described in literature and software engineering standards, works toward a single goal: to produce high-quality software.
  - The users of current software products are surprisingly tolerant to software problems, low-quality software always becomes expensive in the long run.

- While this looks clean enough, we still have to answer the fundamental questions:
  - What kind of software fulfils the high-quality software requirements?
  - How can we ensure that we create high-quality software?
- High quality software views:
  - fulfils customers' needs,
  - follows requirements specifications,
  - gives plenty of functionality,
  - best what one can buy with this price.

- For our purposes, software quality is defined as:
  - Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software (Pressman, Software Engineering, 1997).

- The definition gives us the following points:
  - Requirements specifications define the basis for high-quality software. A software product that does not follow its specifications is not a high-quality product.
  - The theory of software engineering defines the framework where high-quality software must be developed. If used software engineering methods are not within the framework, the result is not a high-quality program.
  - A high-quality software follows both implicit and explicit software requirements.

- One of the implicit software requirement is that is it well tested.
- We cannot test a program completely, but we can create enough test cases to create a good test coverage for the product.
- Why can't we test a program completely?
  - The number of possible test inputs and execution paths through the program can easily exceed our computing capacity.
  - Even if this was possible, each bug fix would require a rerun of the test cases.

- Let's take an example. Myers uses the following program as a self-assessment for his readers' ability to specify adequate testing (Myers, the Art of Software Testing, Wiley, 1979):
  - A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.
- Try to think of the possible test cases before peeking the next slides.

- Although the problem looks simple at first sight, a significant number of test cases must be developed:
  - All input-related problems:
    - too few or too many parameters,
    - parameters are not integers,
    - negative or zero value parameters,
    - parameters with +-sign and/or spaces,
    - integer value overflow problems,
    - etc.

  - Considering that the input values are correct integers:
    - test that the three parameters actually form a triangle,
    - test that the program results a scalene triangle if and only if none of the parameters are equal,
    - test that the program results a isosceles triangle if and only if exactly two of the parameters are equal,
    - test that the program results an equilateral triangle if and only if all three parameters are equal.

  - Output-related problems:
    - test that the output message is correct (no spelling errors in any of the messages etc.),
    - test indirect output (unix pipes for instance),
    - test that output is correct in localised environments (Chinese characters etc).
    - etc.
- Conclusion: Even in such a small program as this, the number of test cases grows very large.
- Corollary: Testing is not as straightforward as it might first sound like.

1.1. Testing as a verification and validation tool

- We define software testing as follows:
  - Testing is a managed activity that systematically by using theoretically accepted testing methods tries to reduce the number of defects in a software product.
- Thus, we consider testing to be
  - theoretically sound,
  - managed, and
  - systematic.

- Testing is a tool for assuring that the tested product fulfils high-quality software requirements. It is used in Quality assurance:
  - Quality assurance: (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured.

- The quality assurance activities can be categorised into verification and validation.
  - Verification refers to the set of activities that ensure that the software correctly implements a specific function.
  - Validation encompasses the set of activities that ensure that the software is traceable to customers' requirements and users' needs.
- Usually this is stated as follows:
  - Verification: Are we building the product right?
  - Validation: Are we building the right product?

- Testing is not the only tool for verification and validation.
  - *Inspections* are used to verify a small piece of documentation or software code. An inspection group gathers into an inspection meeting to review the inspected component. All found problems are written down but their possible solutions are not discussed in the meeting.
  - *Simulation* is used especially in validation. A finished software is put into a simulated environment to check for possible defects. This often saves both time and money.

- More verification and validation tools:
  - *Modeling* belongs to all high-quality software creation. Several models of various abstraction levels can be used in validation. A most common example of this is the use of Universal Modeling Language UML.
  - *Formal methods* are used whenever we have a mathematical model of the product. With mathematical formalism it is possible both to automatically create a frame for the software product and to prove its correctness.
    - Unfortunately this approach is suitable for only a small subset of software products.

1.2. Principles of software testing

- Why testing? Why *systematic* testing?
  - According to Beizer in "Software Testing Techniques", a professionally produced commercial software system contains 1-3 defects per 100 executable statements *when the software engineering group has declared it error-free*.

  - Also Beizer in "Software System Testing and Quality Assurance" reported his private bug rate - how many mistakes he made in designing and coding a program - as *1.5 errors per executable statement.* This includes all mistakes, including typing errors.
  - While most programmers catch and fix more than 99% of their mistakes before releasing a program for testing, there is still the missing 1% left.
  - Human work is needed in test case design, implementation, management, and analysis.

  - Common estimates of the cost of finding and fixing errors in programs range from 40% to 80% of the total development cost.
  - The average debugging effort is about 12 hours of working time for a single defect, testing is one of the most costly phases in a software engineering process.
- Corollary: Companies don't spend fortunes to testing and maintenance to "verify that a program works." They spend the money because the program does not work - it has bugs which need to be found.

- Testing is a process of executing or analysing a software program, its structure or documentation with the intent of finding an error. Thus, testing is not restricted to the software code.
- A good test case is one that has a high probability of finding an as-yet undiscovered error. Thus, test cases must cover the complete product.

- A successful test is one that uncovers an as-yet undiscovered error. Thus, testing is destructive behaviour since its intent is to demolish an existing product.
- Testing cannot show the absence of defects. It can only show that software errors are present. Thus, testing cannot prove that a program is correct.
- Testing is by itself a process that must be systematically managed. Thus, testing is part of software process management.

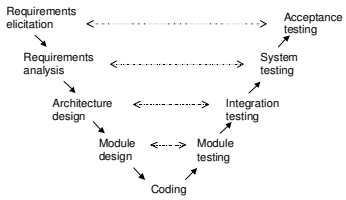
- Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing.
  - All tests should be traceable to implicit and explicit customer requirements.
  - Tests should be planned before testing begins.
  - 80% of all errors uncovered during testing will likely be traceable to 20% of all program modules.

- Basic principles continue.
  - Testing should begin "in the small" and progress toward testing "in the large".
  - Exhaustive testing is not possible.
  - To be most effective, testing should be conducted by an independent third party.
- A software engineer should design a computer program that is easy to test.
  - Fortunately a program that is designed using generally accepted software design and analysis methods is also easy to test.

1.3. The V-model of testing

- Like in other steps in software management processes, also testing needs to be planned and conducted in steps.
- Usually the overall organisation of testing follows the *V-model.*
- The model integrates testing and construction by showing how the testing levels verify the constructive phases.

***The "V" model of software testing (Paakki, Software Testing, 2000)*:**



Specification *Planning* *Testing level* (*phase*)

- The testing process is organised in levels according to the V-model.
- The levels reflect the different viewpoints of testing on different levels of detail.
- Using the V-model integrates testing better to the complete software engineering process. As such it helps to manage testing.
- The presented V-model is not the only one. In fact most companies have their own variations of the model.

- The presented V-model testing levels are module testing, integration testing, system testing, and acceptance testing.
  - In module testing, both module structure and functionality are tested. The structure testing is often called *white-box testing* since source code is visible in it. The functionality testing, on the other hand, is called *black-box testing,* since in it the internal structure of the module is not visible - only its definition is known.
    - Module testing is mostly white-box testing.

  - In integration testing the objective is to verify that modules co-operate correctly. This is achieved by testing both interfaces and global relationships between integrated modules.
    - Integration testing is mostly black-box testing, although also white-box testing can be used to verify module relationships (for instance global variables).
  - At the system testing level, the whole system including hardware, software, and external programs, is tested under the assumption that individual elements are already tested.
    - System testing is completely black-box testing. No source code should be visible anywhere.

  - Acceptance testing is the final testing stage before taking the system into full operation. In acceptance testing customers and users check that the system fulfils their actual needs. Acceptance testing is first conducted in laboratory conditions (so called *alpha-testing)* and finally in a real environment (so called *beta-testing).*
    - Acceptance testing is completely black-box testing as was system testing. Since these testing phases are very similar, they are often combined.

2. Software quality and errors

- The primary task of testing is to find and report errors.
- The result of testing is hopefully improvement in product quality.
- A program's quality depends on:
  - the features that make the customer want to use the program, and
  - the flaws that make the customer wish he'd bought something else.

- Since this material is about testing, we will concentrate on the flaw part of quality: How will we know a flaw when we find it?
  - Thus, what constitutes an error?
- A software error is a mismatch between the program and its specification?
  - Not really. Bad specification leads to bad programs, and implicit requirements exist outside the specification.

- A software error is present when the program does not do what its end user reasonably expects it to do?
  - Better. However, since different users have different expectations, what constitutes a software error also varies. One man's cheese is other man's rotten milk.
- Thus, there can never be an absolute definition for software errors. The extent to which a program has faults is measured by the extent to which it fails to be useful.

- Kaner et al. describes 13 major software error categories in "Testing Computer Software":
  - User interface errors:
    - missing functionality,
    - missing (on-line) documentation,
    - bad command structure,
    - missing user commands,
    - bad performance,
    - badly designed output.

  - Error handling:
    - failure to anticipate the possibility of errors and protect against them,
    - failure to notice error conditions,
    - failure to deal with a detected error in a reasonable way.
  - Boundary-related errors:
    - numeric boundaries: very small numbers, very large numbers, number near zero;
    - program use boundaries: first time, second time, a very large time, after a new document etc;
    - memory configuration errors: too little memory, too much memory (yes, it's possible).

  - Calculation errors:
    - formula errors,
    - precision errors,
    - rounding and truncation errors,
    - calculation algorithm errors.
  - Initial and later states:
    - program fails in the first execution round,
    - program fails at the second execution round,
    - back-up and undo processes are flawed.
  - Control flow errors:
    - wrong execution order,
    - wrong condition branch.

  - Errors in handling or interpreting data:
    - misinterpreted data,
    - missed latest changes during data exchange.
  - Race conditions:
    - program expects two events A and B to occur in order AB while also order BA is possible (while very rare).
  - Load conditions:
    - program fails under a high volume (much work over a long period),
    - program fails under a high stress (maximum load at one time).

  - Hardware:
    - program sends bad data to devices,
    - program ignores device status information,
    - program doesn't recognise hardware failures.
  - Source and version control:
    - old incompatible versions of modules or libraries used in a new program,
    - incorrect copyright messages, sign-on screens, or version numbers,
    - incorrect platform information.
  - Documentation:
    - poor documentation,
    - incomplete documentation.

– Testing errors:
  - errors in test code,
  - errors in test data,
  - misunderstood program functionality in tests,
  - incomplete test coverage.
- Recognising these error classes gives us the first basis for error detection. A more detailed analyse is of course needed for each tested product.
- Errors do vary. Some are inconvenient. Others cause loss of human lives.

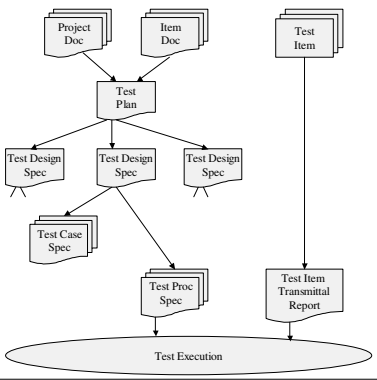
3. Testing management

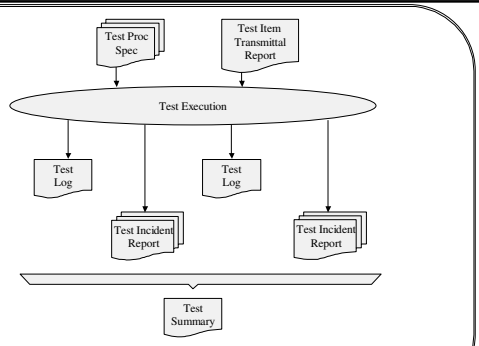
- Testing can be seen as a process by itself. It needs to be planned, executed according to the plan, documented, and reported.
- A well planned and managed testing process simplifies test procedures, helps to create high-quality software, and also helps to repeat tests when necessary.

- The following documents can be used for managing the testing process (Source: IEE829 Standard for Software Test Documentation). The most important documents are in bold.
  – **Test plan** prescribes the scope, approach, resources, and schedule of the testing activities.
  – Test design specification specifies the refinements of the test approach and identifies the features to be tested by the design and its associated tests.

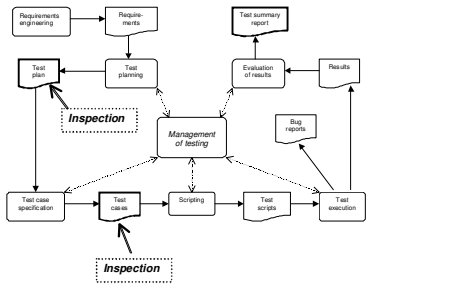
– **Test-case specification** defines a test case identified by a test-design specification.
– Test-procedure specification specifies the steps for executing a set of test cases.
– Test-item transmittal report identifies the test items being transmitted for testing, including the person responsible for each item, its physical location, and its status.
– Test log provides a chronological record of relevant details about the execution of tests.

– **Test-incident report** (or a **Bug report)** documents any event that occurs during the testing process which requires investigation.
– **Test-summary report** summarises the results of the designated activities and provides evaluations based on these results.
- The list must be seen as a guideline from which each organisation can specify and tailor its testing documentation practices.
- The document relationships can be seen in the next two slides.





- The test process model may vary between testing groups.
- The V-model defines the general steps in testing. Each V-model test phase has private process structure that may be linear or iterative.
- One possible test process model as presented by Paakki is in the next slide. It is a minimal model in the sense that it includes only the most necessary steps.



- Next to the test process management, also testing staff must be managed. There are four major ways to share testing responsibilities (Paakki, 2000):
  - 1. ***Testing is each person's responsibility***: the product developers are also responsible for testing their own code. Drawback: testing must be done independently; programmers are too biased by their own solutions and blind to their errors.

  - 2. ***Testing is each group's responsibility***: the product developers within the project group test each other's code. Microsoft: "joint ownership and responsibility". Drawback: usually development efforts finally dominate over testing efforts.
  - 3. ***Testing is performed by a dedicated resource***: there is a person in the project group who only concentrates on testing. Drawback: a person might be nominated who is less effective in development (and probably in testing as well).

  - 4. ***Testing is performed by a test organization***: testing is a component of quality assurance with a dedicated team. Drawback: additional organizations and processes.
- Some testing tasks are among the most mechanical tasks in a software engineering process (test execution from someone else's scripts, for instance). It is important to ensure that the testing staff does not wear out in testing.

4. White-box testing

- *White-box* or *Glass-box testing* is a testing phase where source code is visible.
- In white-box testing, the testing group uses their understanding and access to the source code to develop test cases.
- White-box testing is the kind of testing that is best suited to do once a module or module integration is complete.

- In white-box testing we test *program structure*. We traverse through possible paths, test conditions, loops, and module parameters.
- In the other testing type, *black-box testing*, we test *program functionality*. Program functionality is tested from outside, which is how the customer will work with it.
- Both white-box and black-box testing are needed. White-box testing finds low-level bugs, black-box finds high-level bugs.

- White-box testing provides the following benefits:
  - Focused testing: White-box testing is especially suitable for module-level testing. It's much easier to test an individual module with white-box methods than with black-box methods.
  - Testing coverage: The tester can find out which parts of the program are exercised by any test. White-box testing can tell which lines of code, which branches, or which paths haven't yet been tested, and she can add tests that she knows will cover areas not yet touched.

  - Control flow: The programmer can add test-specific code so that the program constantly reports of what it is doing. The tester can use this code, or run a *debugger* to track the order in which lines of code are executed. The trouble points can be examined and reported immediately.
  - Data integrity: The tester can see which parts of the program modify any item or data. By tracking a data item through the system, she can spot data manipulation by inappropriate modules.

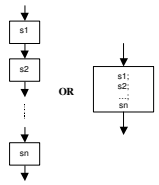
  - Internal boundaries: The tester can see internal boundaries in the code that are completely invisible to black-box testers. For instance, some programs store data into a temporary buffer if too much comes in too quickly. This is visible only at white-box testing.
  - Algorithm-specific testing: Some test cases can be derived directly from the used algorithms. For example, there are many ways to invert a matrix, and well understood ways to miscalculate the result. The programmer can apply standard numerical analysis techniques to predict the results.

- White-box testing methods should always be present in the programming process. The earlier white-box testing is exercised, the earlier low-level bugs are found.
- In an optimal case in the testing process the testers could concentrate on black-box testing. In practice this is not possible, and testers should at least be familiar with white-box testing methods.
- White-box testing can be at least partially automated with appropriate testing tools.

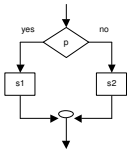
- The principle in white-box testing is to select test cases such that they force the program to move along different kinds of *paths* during its execution.
- A path is one flow of control logic from the beginning of the tested entity to the end of the entity during execution.
- When the tested entity is a program module, a path starts from the module start and ends to one of the exit points in the module.

- The criterion for selecting the execution paths depends on the chosen white-box method. However, all methods are based on the analysis of the control flow of the program.
- The control flow is usually presented as a *flow graph*. The nodes in a flow graph present program statements, and edges present control transfer between the statements.

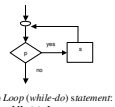
***Flow-graph structures*:**



(a) *Statement sequence*:
s1; s2; …; sn

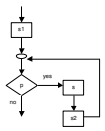


(b) *Conditional* (*if*) *statement*:
if (p) **then** s1 **else** s2

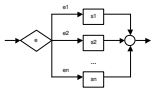


(c) *Loop* (*while-do*) *statement*:
while (p) **do** s

(d) *Loop* (*do-while*) *statement*:
do s **while** (p)



(e) *Iterative* (*for*) *statement*:
for (s1; p; s2) **do** s



(f) *Switch* (*case*) *statement*:
switch (e) {**case** e1: s1;
case e2: s2; …;

4.1. Coverage

- A white-box testing strategy is designed and measured in terms of *coverage*. It defines how well a module is tested with respect of the chosen test policy.
- For the sake of simplicity we consider a tested unit to be a module.
- Coverage is defined with a *coverage criterion*. It indicates how extensively the module has been tested with given test cases.

- Coverage of a program is quantitative. We can calculate the percentage of coverage in a module. This depends on the chosen coverage criterion.
- Before we cover various coverage criteria we need an informal definition for an execution path:
  - An *execution path* in a flow graph is a sequence of nodes starting from the beginning of the module and ending to a module exit point.

- The definition of the execution path is important for all white-box testing.
- Thus, in white-box testing we assume that a flow graph of the tested module is present.
- Although we assume that the tested unit is a module, the methods can be expanded to any program unit where source code is present.

- At least the following coverage criteria has been defined:
  - Statement coverage: A set P of execution paths satisfied the *statement coverage criterion* if and only if for all nodes *n* in the flow graph, there is at least one path *p* in *P* such that *p* contains the node *n*.
    - In other words, statement coverage criterion is fulfilled if test cases force each statement of the tested module to be executed at least once.
    - If a set of execution paths satisfies the statement coverage criterion, we are said to have achieved a complete statement coverage.

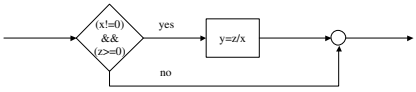
- Branch coverage: A set *P* of execution paths satisfies the *branch coverage criterion* if and only if for all edges *e* in the flow graph, there is at least one path *p* in *P* such that *p* contains the edge *e*.
  - In other words, the test cases must cover each branch for both the true and the false values.
  - The branch coverage criterion is a stronger criterion than the statement coverage criterion. For instance, a test "if (a==1) do_something;" needs a one test case for statement coverage (a=1) but two test cases for branch coverage (a=1, a=0).

- Control coverage: A set *P* of execution paths satisfies the *condition coverage criterion* if and only if for every control node in the flow graph consisting of *atomic predicates* (*c1..cn), ci* yields *true* when evaluated within a path *p1* in *P*, and *ci* yields *false* when evaluated within a path *p2* in *P*, *i = 1,..n*
  - In other words, each atomic test (e.g. a==1) must be evaluated at least once to true, and at least once to false in the test cases.
  - If a branch node has *n* atomic predicates (e.g. a<1 || a >2 || b==0), as many test cases are needed to fulfil condition coverage criterion (tff,ftf,fft).

- Multicondition coverage: A set P of execution paths satisfies the multicondition coverage criterion if and only if for every control node in the flow graph consisting of atomic predicates (c1,…cn), each possible combination of their truth values (true,false) is evaluated within at least one path p in P.
  - In other words, when a branch node has *n* atomic predicates (e.g. a<1 || a>2 || b==0), n*n test cases are needed to fulfil the multicondition coverage criterion (fff,fft,ftf,ftt,tff,tft,ttf,ttt).
  - This is the most complete coverage criterion of practical relevance when testing is driven through exploration of control predicates.

- The essential differences between statement coverage, branch coverage, condition coverage, and multicondition coverage can be summarized by the following conditional statement and its flow graph:

```
if ((x != 0) && (z >=0)) y = z/x;
```



– Complete statement coverage over the module can be reached with a single test: (x=1,z=0).
– Complete branch coverage calls for at least two sets: (x=1,z=0) for the yes-branch, and (x=1,z=-1) for the no-branch.
– Complete condition coverage needs at least two tests: (x=0,z=0) for the atomic combination (false,true) over the control predicate, and (x=1,z=-1) for the atomic combination (true,false).
– Complete multicondition coverage needs four tests: (x=1,z=0), (x=1,z=-1),(x=0,z=0), and (x=0,z=-1). (tt,tf,ft,ff).

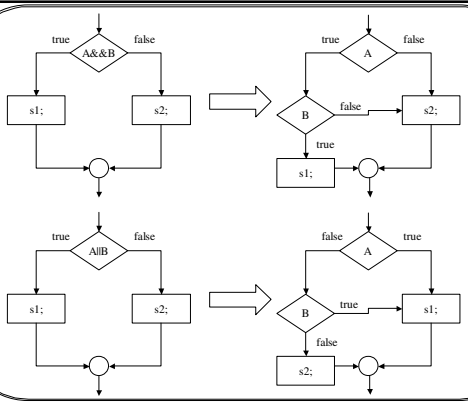
- Next to the branch-based coverage criterion, we can define path-based coverage criterion. Here we define two criteria: path coverage and independent path coverage.
  – Path coverage: A set *P* of execution paths satisfies the *path coverage criterion* if and only if *P* contains all execution paths from the begin-node to the end-node in the flow graph.
    - This is an ultimate white-box coverage criterion.
    - Unfortunately loops make it practically impossible to reach full path coverage for even for relatively simple programs.

– Independent path coverage: A set *P* of execution paths satisfies the *independent path coverage criterion* if and only if *P* contains all the *p+1* independent paths over the flow graph, where *p* is the number of predicate nodes in the graph.
  - A set of paths is called independent if none of them is a linear combination of the others. In other words, a linear path cannot be combined from other linear paths.
  - Independent paths collapse loops into single iterations and therefore dramatically reduce the number of test cases needed to explore the module.

- The independent path coverage is a good criterion for path-based testing. It can be made more complete by breaking tests to atomic predicates with the following simple rules:

  if (A && B) s1; else s2; ==> if (A) if (B) s1; else s2; else s2;

  if (A || B) s1; else s2; ==> if (A) s1; else if (B) s1; else s2;
  – This way independent path coverage criterion comes closer to the multicondition coverage criterion.
  – Similarly multipredicate branches are broken into atomic predicates.



- The set of independent paths can be constructed by starting from the shortest execution path, and incrementally introducing a new path by adding at least one edge that is not yet contained in any path in the set.
  – Sometimes this leads to impossible execution paths. Some fine tuning is then necessary.
- If the tested module has several exit points, a new artificial exit node can be introduced to simplify the test path generation.

- When a particular coverage criterion has been chosen, the test cases that satisfy the criterion must be selected. In principle this is straightforward:
  - 1. Construct the flow graph of the tested module.
  - 2. Choose the minimum number of execution paths that satisfy the criterion, and
  - 3. For each selected path, prepare a test case that activates the execution of the path.
- These tasks can be partially automated.

- Example: Create test cases for the following module using independent path coverage.

```
PROCEDURE average
* This procedure computes the average of 100 or
  fewer numbers that lie between bounding values. It
  also computes the sum and the total number valid.
INTERFACE RETURNS average, total.input,
  total.valid
INTERFACE ACCEPTS value, minimum, maximum
TYPE value [1:100] IS SCALAR ARRAY
TYPE average, total.input, total.valid, minimum,
  maximum, sum IS SCALAR
TYPE i IS INTEGER
```

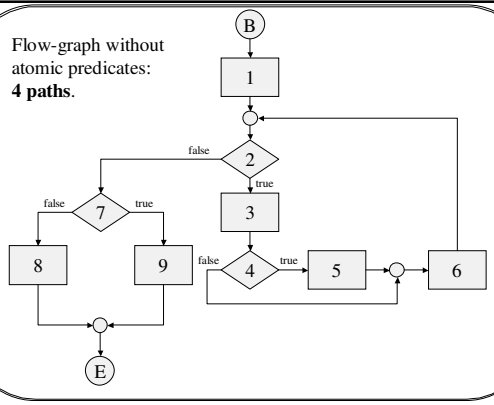


```
* Example continues
i = 1
total.input = total.valid = 0
sum = 0
DO WHILE value[1] <> -999 AND total.input < 100
   increment total.input by 1
   IF value[i] >= minimum AND value[i] <=
  maximum THEN
        increment total.valid by 1
        sum = sum + value[i]
   ENDIF
```



```
* Example continues
   increment i by 1
ENDDO
IF total.valid > 0 THEN
      average = sum / total.valid
   ELSE
      average = -999
ENDIF
END-PROCEDURE average
```


Flow-graph without atomic predicates: **4 paths**.

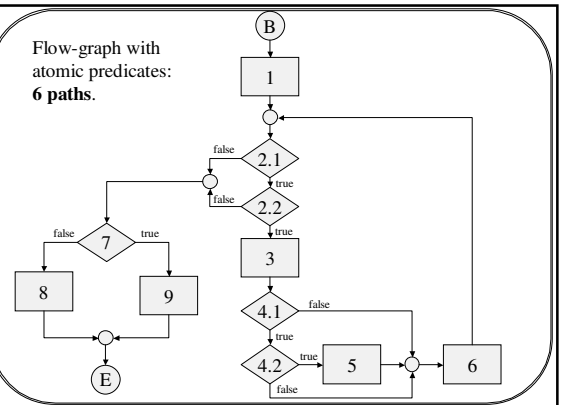


- Flow-graph nodes are as follows:
  - 1: i=1, total.input = total.valid = 0, sum = 0
  - 2: value[i] <> -999 AND total.input < 100
  - 3: increment total.input by 1
  - 4: value[i] >= minimum AND value[i] <= maximum
  - 5: increment total.valid by 1, sum=sum+value[i]
  - 6: increment i by 1
  - 7: total.valid > 0
  - 8: average = -999
  - 9: average = sum / total.valid

- The four independent paths are:
  1) B,1,2,7,8,E
  2) B,1,2,7,9,E
  3) B,1,2,3,4,6,2,...,7,8,E
  4) B,1,2,3,4,5,6,2,...,7,9,E
     (This is a better alternative to path B,1,2,3,4,5,6,2,...,7,**8**,E which is also independent of paths 1-3)
- The test cases could be:
  1) value[0] = -999
  2) Path cannot be tested standalone.
  3) minimum = 1, value[0] = 0, value[1] = -999
  4) minimum = 1, value[0] = 1, value[1] = -999

- The original independent path coverage gives rather poor white-box test cases.
  - The maximum number of data items is never tested.
  - The test against a maximum input value is never tested.
- To get better test cases we must break the non-atomic predicates in branches to atomic predicates. This is done in the next slide.

Flow-graph with atomic predicates: **6 paths**.



- Flow-graph nodes are as follows:
  1: Same as earlier
  2.1: value[i] <> -999
  2.2: total.input < 100
  3: Same as earlier
  4.1: value[i] >= minimum
  4.2: value[i] <= maximum
  5-9: Same as earlier

- Now the six independent paths are:
  1) B,1,2.1,7,8,E
  2) B,1,2.1,7,9,E
  3) B,1,2.1,2.2,...,7,9,E
  4) B,1,2.1,2.2,3,4.1,6,2,...,7,9,E
  5) B,1,2.1,2.2,3,4.1,4.2,6,2,...,7,9,E
  6) B,1,2.1,2.2,3,4.1,4.2,5,6,2,...,7,9,E
- The test cases could be:
  1) value[0] = -999
  2) Cannot be tested standalone
  3) minimum=1,maximum=2,value[0]..value[99]=1
  4) minimum=1,maximum=2,value[0]=1,value[1]=0,value[2]=-999
  5) minimum=1,maximum=2,value[0]=1,value[1]=3,value[2]=-999
  6) minimum=1,maximum=2,value[0]=1, value[1]=2, value[2]=-999

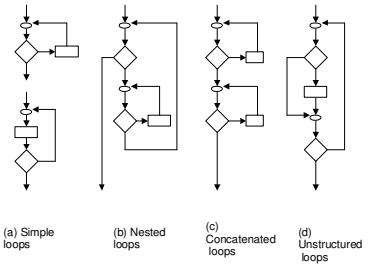
- It is important to note that some independent paths cannot be tested in standalone fashion.
  - The combination of data required to traverse the path cannot be achieved in the normal flow of the program.
- In such cases, these paths are tested as part of another path test.
- Sometimes an independent path must be expanded to achieve the end condition. The path 3 earlier is a typical example of this.

4.2. Loop testing

- One problem of the independent path coverage testing is that it collapses loops into a single iteration when possible. Thus, many loop-related bugs are not discovered by the conventional coverage methods.
- Yet loops are a cornerstone of a vast majority of algorithms. They must be tested with more than the smallest number of test cases that span a particular coverage.

- Loop testing is a white-box technique that focuses on the validity of looping constructs with respect to control flow of the program.
- Four different classes of loops can be defined, each with a testing strategy of its own:
  - Simple loops,
  - nested loops ,
  - concatenated loops, and
  - unstructured loops.

7.2. Loop testing



- Typically loop bugs lurk in corners and attack at boundaries. Thus interesting loop iterations are:
  - 0 rounds (skip the loop entirely)
  - 1 round
  - 2 rounds
  - *m* rounds (a typical value)
  - max-1 rounds
  - max rounds
  - max+1 rounds

- If it is known that the loop has internal boundaries, the boundaries are treated the same way than maximum values:
  - A loop reads a physical file that is made of blocks of 1024 bytes then we do:
    - 0,1,2,typical size,1023,1024,1025,max-1,max,max+1 iterations.
  - A dynamic array grows physically after 256 added data items:
    - 0,1,2,typical size,255,256,257,max-1,max,max+1 it.
  - Sometimes the max value is hardware-dependent, sometimes it's difficult to calculate.

- Simple loops:
  - Each simple loop is analysed with the previously explained passes (0,1,2,typical,etc.)
  - If the number of iterations is fixed or restricted, we may be able to test just a few of the iteration patterns given earlier.

    FOR i=0 to 10 DO ....
- Nested loops:
  - If we extend the simple loop strategy to nested loops, the number of necessary tests will grow geometrically with the level of nesting.

– The geometrical growth soon creates too many test cases to be practical. A two-level or at most a three-level nesting is the maximum for this kind of complete testing. For instance, when we have a nested loop with a known inside boundary (like a file block size) we get the following cases:
  - Two-level nesting needs 10*9+1 = 91 tests.
  - Three-level nesting needs 10*9*9+1=811 tests etc.
  - The extra one case comes from the zero case when all loops are skipped.
  - Combinations (0,0,any) and (0,any,any) are not possible.

– Since the number of test cases soon grows to impractical numbers, the following optimised strategy may be applied:

1) Start at the innermost loop. Set all other loops to 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. Add other tests for out-of-range or excluded values.
3) Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4) Continue until all loops have been tested.

- Concatenated loops:
  – Concatenated loops can be tested using the approach defined in the simple loops if each of the loops is independent of the other.
  – 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.

- Unstructured loops:
  – Whenever possible, this class of loops should be redesigned to reflect the use of well-behaving loops (simple, nested, and concatenated loops).
  – Such restructuring is always possible since nested loops along with condition branches allow any combination of loop iterations.

4.3.Data-flow testing

- The white-box testing methods presented earlier are based on studying the control flow over the program under test.
- Although various path testing coverage criteria are highly effective, they are not sufficient alone.
- The path testing coverage criteria are based on paths. Yet some bugs lurk in data.

- Even if every path is executed once, the program can still have bugs. Here is an example:

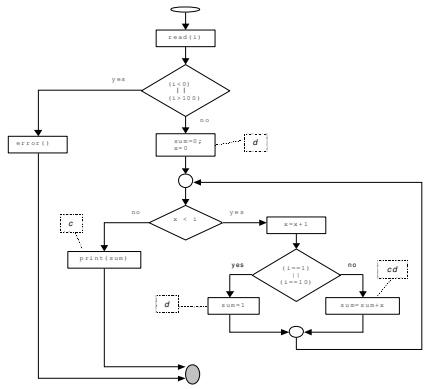
```
a = b/c;
```

  – This works if *c* is nonzero, but what if *c* is 0? The difference between the two cases is a difference in data, not in paths.
- A path is called *error-sensitive* if an error in it *can* be detected when it is exercised. A path is *error-revealing* if the error *always appears* when the path is taken.

- Any path that includes the division *b/c* is error-sensitive since the error depends on the value of *c*.
- Due to this, a still more detailed category of methods, *data-flow testing,* takes additionally into account the things that might happen to data when the module is executed.
- In data-flow testing, the central events related to variables are value assignments (value writes) and value uses (value reads).

- The value use may be in computation or in a control predicate. These cases are also evaluated in data-flow testing.
- Actually the value assignment may be in a control predicate as well, for instance in C-language and in other similar languages.
- A special flow graph called *data flow graph* is generated for each analysed variable. It is then used to define variable-specific test cases.

- A data flow graph is similar to a traditional flow graph, except that it has added syntax for variable presentation. The presentations vary between models, but mostly they show variable definitions and use.
- An example of a data flow graph is in the next slide.



- The data-flow methods are usually complex, relatively difficult to use, and not very good at detecting data errors.
- The data-flow analysis is usually based on a single variable. Thus, each variable needs its own analysis and test cases. This can lead to a very complex testing procedure.
- While lately some progress is being made in data-flow testing, value-dependent error detection is still mostly left for black-box testing.

5. Black-box testing

- Black-box testing focuses on the functional requirements of the software. The program is considered a black box without structure.
- Program logic may be present in black-box analysis, but implementation details are not. The internal details of modules and subsystems are hidden and cannot be studied from outside.

- Black-box testing concentrates on the interfaces of modules and subsystems. It is used to test externally observable functionality and input-output behaviour. It is often called functional testing, while white-box testing is procedural testing.
- Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover different class of errors than white-box methods.

- Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.
- The theory of black-box testing is not as mature as the theory of white-box testing due to lack of mathematical background for functionality.
- Yet black-box testing is the kind of testing that most people consider "real testing".
- Still black-box testing need not to be magic.

- The black-box testing is based on input values. If we could test the program with all possible input values *and* evaluate the results of the tests then we could say that we have a complete black-box testing.
- Unfortunately this would imply
  - all possible valid inputs,
  - all possible invalid inputs,
  - all possible edited inputs,
  - all possible combinations of inputs, and
  - all possible input timings.

- Even with a small input value set the number of possible test cases would grow too large. Due to this we need to prune the input value set.
- The solution for the pruning lies in *equivalence classes*.
  - An equivalence class is a set of input values that cause an equal execution path in the program.
  - Thus, any value in the class represents the behaviour of all values in the class.

- Hence, when we divide the input set into a set of equivalence classes and pick one input value from each class we get a complete black-box testing.
- Unfortunately (again) the chosen equivalence classes depend on the analyser. Two people analysing a program will come up with a different list of equivalence classes.

- Here are a few recommendations for looking for equivalence classes:
  - Look for invalid inputs.
    - If all invalid inputs are processed similarly they create an equivalence class. If some cause different program execution (not just a different error message) they create several equivalence classes.
  - Look for ranges of numbers.
    - A range creates several equivalence classes. Let's say that we accept values 1-99. Then we have
      - a class of values 1-99,
      - a class of values < 1
      - a class of values >99, and
      - a class of non-numbers.

– Look for membership in a group.
  - If an input must belong to a group, one equivalence class includes all members of the group. Another includes everything else.
  - Sometimes it is reasonable to subdivide the classes into smaller subclasses.
– Analyse responses to lists and menus.
  - When the program responds differently to each alternative of a list or a menu, the alternatives are equivalence classes.
  - If several alternatives create a same response (a menu and a hot key for instance) they form a single equivalence class.

– Create time-determined equivalence classes.
  - When timing matters to input values, we get several time-related equivalence classes. Let's say that time *t* is a critical input time. Then we get the following equivalence classes:
    – input values long before time *t*,
    – input values shortly before time *t*,
    – input values at time *t*,
    – input values shortly after time *t*, and
    – input values long after time *t*.
  - Naturally this kind of partitioning is relevant only when time plays a critical role in the system.

– Look for variable groups that must calculate to certain value of range.
  - Sometimes a function determines the equivalence classes. Let's assume that we have a program that accepts three angles of a triangle. Thus we have a function a1+a2+a3 = 180, where a1, a2, and a3 are given angles. Then we have four equivalence classes:
    – The sum of the input values is 180.
    – The sum of the input values is <180.
    – The sum of the input values is >180.
    – One or more of the input values is not a valid number.

– Look for equivalent output events.
  - The equivalence class may be determined by an output event. For instance, if a program drives a plotter that can drive lines up to four inches long, a line might be within the valid range, there might be no line, the program might try to plot a line longer than four inches, or it might try to plot something else altogether.
  - Some difficulty lies in determining what inputs to feed the program to generate these different outputs.
  - Sometimes many classes of inputs should have the same effect, and unless we are know for certain that they are treated similarly in the program they should be separated to different equivalence classes.

– Look for equivalent operating environments.
  - If a program has expected or known hardware limitations, they create equivalence classes. For instance, if a program is stated to work with memory sizes of 64M - 256M, the three equivalence classes include a memory < 64M, a memory in 64M - 256M, and a memory > 256M.
  - Sometimes hardware expectations cause severe failures. For instance, if the program has a race condition it is possible that a slower (or a faster) CPU triggers it and the program fails.
  - All hardware-specific limitations create equivalence classes (where the input value is hardware instead of data).

- Usually we are not dealing with scalar inputs. Instead, we need an *input vector* that defines several input parameters to the program.
- In that case the input domain of the system in a mathematical sense is a cartesian product:

  $D = D1 \times D2 \times ... \times Dn$, where $D1, D2, ..., Dn$ are the input domains of the vector elements.

- Then the equivalence classes for the whole input can be designed as combinations of the classes for the elementary domains.
- For instance, let's assume that a program accepts two integers between 100 and 200.
- For each of the input integers we get the following equivalence classes:
  - valid numbers 100...200
  - integers < 100
  - integers > 200
  - non-integers

- Altogether we get 16 equivalence classes:
  - (x valid, y valid),
  - (x < 100, y valid),
  - (x > 200, y valid),
  - (x non-integer, y valid),
  - (x valid, y < 100),
  - (x valid, y > 200),
  - (x valid, y non-integer),
  - ...
  - (x non-integer, y non-integer).

- As can be noticed, a complete cartesian product even over relatively simple domains soon leads to an explosion in the number of equivalence classes.
- Also important is to notice that almost all the cases are of illegal combinations.
- One way to prune the equivalence class set is to take a few cases for closer inspection.
- This sort of leads to equivalence classes of equivalence classes.

5.1. Class representative selection

- In an optimal case the equivalence class partition is sound and complete. No matter which representative we elect the result of the corresponding test remains the same.

- In the formal sense, "identical" behaviour is not a solvable problem. So, without executing a program it is usually not possible to know whether two different input values are actually processed in the same way.
- Since we cannot ensure that our partition is perfect, it is bad testing practice to pick just a single value from each equivalence class.
- Instead we take two or three values from each class and run the tests with them.

- If the chosen input values result different test results then we know that the equivalence class is too large: each member in an equivalence class should generate the same test response.
- In such a case the equivalence class is partitioned into two smaller cases such that the differing inputs are no longer in the same class.

- When a testing strategy is based on equivalence classes, the best input values reside close to class boundaries.
- Values that most likely will reveal bugs in the tested software are those that reside at a boundary line between several equivalence classes.
- For instance, if we our program accepts integers between 10...100, the boundary values 10 and 100 reveal lower and higher relational operation bugs.

- If we have picked any other values from the equivalence class then the relational operators would not have been tested.
- Also when a program fails a test that uses any value from an equivalence class, it will fail the same test when using a boundary value. The reverse is not true as stated earlier.
- A special case in the boundary selection is value 0 which should be selected no matter where it resides in its equivalence class.

- Actually quite often an equivalence class including value 0 is divided into three equivalence classes: values < 0, value 0, and values > 0.
- If we are not at all certain of the equivalence classes, it is often better to consider the input domain as a single class and pick random input values from there.
- This leads to better testing coverage than picking random equivalence classes and their representatives.

5.2. Control and data flow testing

- So far we have assumed that we have absolutely no knowledge of the design of the tested program.
- Usually, however, we have information of the control flow in data flow diagrams, state transition diagrams, and modeling language presentations.
- This information should be used in equivalence class generation.

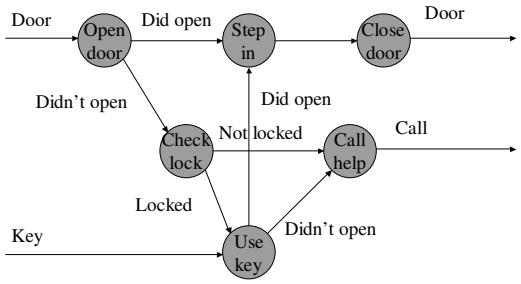
- When the analysis includes *a data flow diagram*, we have information about the control logic of the program.
- Often this information includes at least defined logical processes, data flowing between the processes, and conditions presenting when data flows to processes.
- This information is already analysed and presents some or all of the requirements for the program.

- The data flow analysis has the following steps:
  - 1. Examine the requirements and compare them to the data flow model generated. The model and requirements should be equal: model is a lower abstraction of the requirements.
  - 2. If the model and requirements are not equal, the model must be updated to be a presentation of the requirements. However, if the model represents only a subset of the requirements then the model need not to be updated. The test cases will just reflect the modelled subsystem.

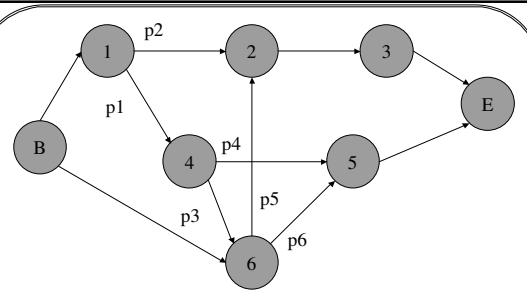
– 3. Number the nodes uniquely. The numbers will be used as node names later.
– 4. For each of the data flow branches identify the branch condition and add it to the model.
– 5. Create a start node from where all incoming data flows start, and an end node to where all outgoing data flows end.
– 6. Break all composite predicates to atomic predicates. Add new processes and flows to reflect the changed control flow logic.

– 7. Now you should have a black-box based *control flow graph* of the tested system. Generate test cases based on independent path coverage criteria, as presented in white-box testing. The test inputs are the incoming flows. The test outputs are the outgoing flows.
– 8. Each generated test case represents an equivalence class. Identify the class boundaries and add new classes when necessary.

- As an example let's take a look at a problem of opening a door which may or may not be locked:



- In the data flow diagram the door is opened. If the opening is successful the opener steps in and closes the door behind her. If something goes wrong, the door is first tried to unlock with a key, and if this does not work help is called to open the door.
- In the generated black-box control-flow graph we have numbered the nodes, added an entry and an exit node, and identified the branch testes p1..p6.



Independent paths:
B,1,2,3,E (door not locked and opens)
B,1,4,5,E (door not locked but doesn't open)
B,6,2,3,E (door is locked and opens with a key)
B,6,5,E (door is locked and doesn't open with a key)

- Our example is a bit simplistic in a sense that it presents detailed control.
- A typical data flow diagram concentrates more on the flow of data (as the name suggests). In such cases the first step in equivalence class generation is to refine the diagrams to such that they represent control flow with data flow.
- This can be achieved with help from requirements specifications and user manuals.

5.3. Outcome prediction

- Now that we have chosen our equivalence classes and run our tests, we still need to analyse the outcome of the tests. This is not as straightforward as it first sounds. We need *an oracle.*
- The first idea in mind is to play a manual computer and try to predict the outcome of a specific input. It seldom works.

- First of all, it can be very hard work if the control logic of the program is complex.
- Second, while simulation is a good way for validation, humans are not very good at simulating computers. The result is extremely error-prone.
- Third, when we do manual prediction, we add a new considerable source of errors to the testing procedure. We are likelier to make an error in the manual outcome than the programmer is in programming.

- Beizer lists in his book "Black-Box Testing" (Beizer, 1995) several more attractive alternatives:
  - Existing tests: Most testers and programmers work on modifications to a base of existing software. When most tests do not change from release to release, they can and should be re-used.
  - Old program: A major rewrite need not to imply that its outputs are completely different. The old program is an excellent oracle when it is available and its specification is valid.

  - Previous version: Even if the tested code has been rewritten, the previous version of the program will often have the correct outcome for most test paths.
  - Prototypes: A prototype is an excellent oracle. Good prototypes do not lack functionality. The reason that they are not operationally useful is that they may be too slow, too big, not maintainable enough, or can't run in the target environment.

  - Model programs: A model program is a simplified prototype that includes only functionality that is directly related to the application. Such a program is considerably easier to build than a complete program with error checking, user interfaces, and data communication.
  - Forced easy cases: Sometimes it is possible to select input values from an equivalence class such that the outcome is trivial to calculate. Such values are excellent input candidates since they simplify output analysis.

  - The actual program: The actual program can be used as an oracle as long as program requirements are well understood and the program output is carefully verified. Usually it is easier to verify the correctness of an outcome than it is to calculate the outcome by manually simulating the computer. When the program includes information about its intermediate states, the verification procedure becomes significantly easier.

5.4. Main differences between black-box and white-box testing

- white-box: source code is needed,
- black-box: test cases are obtained directly by design,
- white-box: test cases obtained by separate code analysis,
- black-box: less systematic and "formal", more ad-hoc and based on intuition ==> technically easier but more risky,
- white-box: strong theoretical background, automation possibilities (coverage),
- black-box: no standard notion for internal testing quality (coverage), and
- black-box: more common in industry.

6. Module testing

- As mentioned in the V-model, we have at least the following steps in a testing process:
  - module testing, where each unit is tested separately,
  - integration testing, where units are integrated and unit interfaces and dependencies are tested,
  - system testing, where the whole system including hardware and other software components are tested together, and

  - acceptance testing, where customers verify that the created software product fulfils its requirements.
- The methods presented in the previous two chapters, white-box testing and black-box testing, are used in the four test procedure steps.
- The first step, module testing (or unit testing), focuses verification effort on the smallest unit of the software.
- Usually the tested unit is a module.

- Module testing is mostly white-box testing. However, also black-box testing is useful especially when we are testing control modules.
- The areas to consider in module testing are
  - interfaces,
  - local data structures,
  - path and loop testing,
  - error conditions,
  - and boundary conditions.

6.1. Module interface testing

- Tests of data flow across a module interface are required before any other test is initiated.
- If data do not enter and exit properly, all other tests are useless.
- Myers presents in his book (Myers, The Art of Software Testing, 1979) a checklist for interface tests.
  - 1. Number of input parameters equal to number of arguments?

  - 2. Parameter and argument attributes match?
  - 3. Parameter and argument units systems match?
  - 4. Number of arguments transmitted to called modules equal to number of parameters?
  - 5. Attributes of arguments transmitted to called modules equal to attributes of parameters?
  - 6. Unit system of arguments transmitted to called modules equal to unit system of parameters?

- 7. Number attributes and order of arguments to built-in functions correct?
- 8. Any references to parameters not associated with current point of entry?
- 9. Input-only arguments altered?
- 10. Global variable definitions consistent across modules?
- 11. Constraints passed as arguments?

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

- 1. File attributes correct?
- 2. OPEN/CLOSE statements correct?
- 3. Format specification matches I/O statement?
- 4. Buffer size matches record size?
- 5. Files opened before use?
- 6. End-of-file conditions handled?
- 7. I/O errors handled?
- 8. Any textual errors in output information?

- When all these steps are properly executed, unit integration becomes much simpler and easier to manage.

6.2 Local data structures

- The local data structures for a module are a common source of errors. According to Pressman (1997), test cases should be designed to uncover errors in the following categories:
  - 1. improper or inconsistent typing.
  - 2. erroneous initialisation or default values.
  - 3. incorrect (misspelled or truncated) variable names.

- 4. inconsistent data types.
- 5. underflow, overflow, and addressing exceptions.

- In addition to local data structures, the impact of global data on a module should be ascertained during module testing.
- If the effects of accessing global data from a module cannot be predicted, it may be a sign of bad design, and it is absolutely a risk for nasty unpredictable errors.

6.3. Path and loop testing

- Selective testing of execution paths is an essential task during module testing.
- Independent path coverage and loop testing are effective techniques for uncovering a broad array of path errors.
- Path and loop testing are good at finding errors in predicates since a predicate implies a branch in software code.

- Whenever possible, path and loop testing should include test cases for testing error-sensitive paths (that is, paths where an error depends on the value of the test input).
- Data flow coverage methods can be used for error-sensitive path testing. However, when modules are independent, well designed, and small enough, the error-sensitive sentences are easy to isolate from the code.
- Then new test cases can be created to especially test the error-sensitive paths.

6.4. Error condition testing

- Error conditions are often the least tested areas in software. Yet high quality software should include clear and covered error handling including
  - error detection,
  - error recovery,
  - clean exit (if recovery is not possible), and
  - easily testable error condition paths.

- Among the potential errors that should be tested when error handling is evaluated are:
  - 1. Error description is unintelligible
  - 2. Error noted does not correspond to error encountered.
  - 3. Error condition cuases system intervention prior to error handling.
  - 4. Exception-condition processing is incorrect.
  - 5. Error description does not provide enough information to assist in the location of the cause of the error.

6.5. Boundary testing

- Boundary testing is the last task of the module testing step.
- Since errors lurk in boundaries, this is step must be executed with extreme care.
- We have already covered various boundary conditions in loop testing and black-box equivalence classes.
- At least the following must be tested:

  - 1. Parameter boundaries.
    - The minimum and maximum values of a parameter are good test inputs. Also value zero is a good one. This is black-box testing.
  - 2. Loop boundaries.
    - As mentioned in loop testing, 0, 1, n-1, n, and n+1 iterations must be tested for each loop. Here *n* is the maximum number of passes in the loop.
  - 3. Data structure boundaries.
    - Data structures are fruitful in detecting boundary errors. For instance, dynamic data structure should be tested with values that make the data structure grow (or shrink). This is mostly black-box testing.

6.6. Module test procedures

- Module testing is normally considered as a step in program coding.
- This is true in a sense that module testing should be done as soon as the module is finished.
- Nevertheless it is still a testing step which needs both white-box and black-box testing methods.

- Because a module is not a standalone program, *driver* and/or *stub* software must be developed for each module test.
- In most applications a driver is a procedure that accepts test case data, passes data to the tested module, gathers result information, and forwards the result information to an analysing procedure which returns information of the outcome of the test.
- Stubs serve to replace modules that are called by the module to be tested.

- A stub uses the caller module's interface, does minimal data manipulation, prints a verification of entry, and returns.
- Drivers and stubs represent overhead. They must be present in testing but they are not included in the final product.
- Due to this drivers should be designed such that they can be used on several modules.
- Stubs should be designed such that they do the absolutely minimum tasks necessary.

7. Integration testing

- The next step after module testing is integration testing. In it the modules are systematically added into an evolving system.
- Integration testing tests module interfaces and co-operation. Two standard criteria of modularity has great effect on these areas: *module cohesion* and *module coupling*.

- Module cohesion describes how well a module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.
- In other words, a high cohesive module does just one thing well and with minimum interaction with other modules.
- The exact cohesive level of a module is not important. Rather it is important to strive for high cohesion and recognise low cohesion.

- Module coupling is a measure of interconnection among modules in a program structure.
- Coupling depends on the interface complexity between modules, what data pass across the interface, and how global software entities are shared between them.
- The lower is a module coupling between two modules, the less they interact with each other, and the easier they are to integrate.

- The various coupling levels from lowest to highest are as follows:
  - No coupling: two modules are totally unrelated.
  - Data coupling: module interface uses variables.
  - Stamp coupling: module interface uses a portion of a data structure.
  - Control coupling: module interface passes a control flag (a variable that controls decisions in the accessed module).
  - External coupling: modules are tied to an environment external to software (for example, one module reads a file, another writes it).

  - Common coupling: modules are tied to global variables.
  - Content coupling: one module makes use of data or control information maintained within the boundary of another module, or branches are made into the middle of a module.
- All coupling levels excluding content coupling have their uses. The important thing is to strive for the lowest possible coupling.

- When all the modules are linked and tested as a whole, we are doing *big-bang* integration. This is the most common integration policy in small programs.
- Big-bang integration is a simple solution which requires no extra effort because neither drivers nor stubs are needed.
- The problems arise when failures occur. The testers probably have no idea in which module the actual error might lurk since the tested unit is a monolithic program.

- Two more elegant approaches to integration testing are *top-down integration* and *bottom-up integration.*
- In top-down integration the highest level control module (main loop) is integrated to its called modules. The other modules in the next level are replaced with stubs.
- When co-operation between these modules has been tested, the stubs are replaced with the next level modules whose called modules are again replaced with stubs.

- The previous procedure is repeated until all modules are integrated to the system; thus the name top-down integration.
- Top-down integration is suitable to situations where program logic needs first-level testing. It is also suitable for situations when we want to have an immediately executable program.
- The drawback of top-down integration is the need of stubs. Some of the stubs may become very complex.

- In bottom-up integration the lowest level modules are integrated into *clusters*. Each cluster fulfils one subtask of the program.
- Clusters are tested separately. Thus, each cluster needs a driver that feeds it test cases.
- When all clusters are tested, their drivers are replaced with next level control modules. This generates new clusters that are little larger than earlier ones.
- The procedure is repeated until all clusters are combined into a single program.

- Bottom-up testing is suitable to software where program functionality needs more testing than program logic. Problems in logic reflect down to the processing clusters which may nullify the results of already made tests.
- *Regression testing* is needed to ensure that earlier integration testing is still relevant.
- The good side of bottom-up testing is the lack of stubs. Next to drivers all needed code is directly related to the program.

- Bottom-up pros and cons (Paakki, 2000):
  - \+ Drivers are easier to write than stubs.
  - \+ The approach provides a greater opportunity for parallelism than the other approaches; that is, there can be several teams testing different subsystems at the same time.
  - \+ The approach is effective for detecting detailed design or coding errors early enough.
  - \+ The approach detects critical performance flaws that are generally associated with low-level modules.
  - \+ The approach supports the modern software engineering paradigms based on classes and objects, or reusable stand-alone components.
  - – A prototype of the system is not available for (user) testing until very end.
  - – The approach is not as effective for detecting structural design flaws of large (top-level) scale.

7.1. Regression testing

- Each time a new module is added as part of integration testing, the software changes.
  - New data flow paths are established,
  - new I/O may occur, and
  - new control logic is invoked.
- The changes may cause problems with functions that previously worked flawlessly.

- *Regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- In a boarder sense, regression testing is the activity that helps to ensure that changes in software (due to testing or maintenance) do not introduce unintended behaviour or additional errors.

- The regression test suite (the subset of tests to be executed) contains three different classes of test cases:
  - A representative sample of tests that will exercise all software functions.
  - Additional tests that focus on software functions that are likely to be affected by the change.
  - Tests that focus on the software components that have been changed.

- As integration testing proceeds, the number of regression tests can grow quite large.
- The regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.
- It is impractical and inefficient to re-execute every test for every program function once a change has occurred.
- *Automated tests* are especially useful in regression testing.

8. Other testing phases

- When module and integration testing are finished, the software is completely assembled as a package.
- The next step in testing is then *System testing*. It is a series of different tests whose primary purpose is to fully exercise the computer-based system including all software and hardware components.

- System testing may include at least the following testing series:
  - Recovery testing. A system test that forces the software to fail in various of ways and verifies that recovery is properly performed.
  - Security testing. A system test that attempts to verify that protection mechanisms built into a system will in fact protect it from improper penetration.
  - Stress testing. A system test that executes the system in a manner that demands resources in abnormal quantity, frequency, or volume.

- When system testing is over, the final step in the testing process is *Acceptance testing*. In it customers and users verify that the product fulfils their actual needs and follows requirements.
- Acceptance testing includes at least the following testing steps:
  - Validation testing. Software validation is achieved through series of black-box tests that demonstrate conformity with requirements.

  - Configuration review. The review ensures that all elements of the software configuration have been properly developed, are catalogued, and have the necessary detail to support the maintenance phase of the software life cycle.
  - Alpha testing. The software is used by end-users in a natural setting in laboratory conditions. The development team is recording errors and usage problems.
  - Beta testing. The software is used by end-users in an environment that cannot be controlled by the development team.

9. Testing for specialised environments

- As computer software has become more complex, the need for specialised testing approaches has also grown.
- In this section we will briefly discuss user-interface testing, testing client-server architectures, testing documentation and help facilities, and testing for real-time systems.

9.1. User interface testing

- Since modern graphical user interfaces (GUIs) have the same look and feel, a series of standard tests can be derived.
- The following areas can serve as a guideline for creating a series of generic tests for GUIs (Pressman, 1997):
  - windows-specific tests,
  - pull-down menu and mouse specific tests, and
  - data entry tests.

- Here is a checklist for windows:
  - Will the window open properly based on related typed or menu-based commands?
  - Can the window be resized, moved, and scrolled?
  - Is all data content contained within the window properly addressable with a mouse, function keys, directional arrows, and keyboard?
  - Does the window properly regenerate when it is overwritten or resized, and then recalled?
  - Are window-related functions available when needed?

- A checklist for pull-down menus:
  - Is the appropriate menu bar displayed in the appropriate context?
  - Does the application menu bar display system related features?
  - Do pull-down operations work properly?
  - Do breakaway menus, palettes and tool bars work properly?
  - Are all menu functions and pull-down subfunctions properly listed?
  - Is it possible to invoke each menu function using its alternative text-based command?

- A checklist for mouse operations:
  - Are all menu functions properly addressable by the mouse?
  - Are mouse operations properly recognised throughout the interactive context?
  - If multiple clicks are required, are they properly recognised in context?
  - Do the cursor, processing indicator, and pointer properly change as different operations are invoked?
  - Is help available and is it context sensitive?

- A checklist for data entry:
  - Is alphanumeric data entry properly echoed and input to the system?
  - Do graphical modes of data entry work properly?
  - Is invalid data properly recognised?
  - Are data input messages intelligible?
- In addition to above guidelines, finite state modeling may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI.

9.2. Testing documentation

- Errors in documentation can be as devastating to the acceptance of the program as errors in data or source code.
- For this reason, documentation should be a meaningful part of every software test plan.
- Documentation testing can be approached in two phases: via inspections and in a live test.

- Inspections may be used to test editorial clarity:
  - Software functionality is fully covered in the manual.
  - No typing errors are present.
  - Etc.
- Live tests uses the documentation in conjunction with the use of the actual program.
  - Program usage is tracked through the documentation.

- The following checklist may be used in documentation testing:
  - Does the documentation accurately describe how to accomplish each mode of use?
  - Is the description of each interaction sequence accurate?
  - Are examples accurate?
  - Are terminology, menu descriptions, and system responses consistent with the actual program?
  - Is it relatively easy to locate guidance in the documentation?

  - Can troubleshooting be accomplished easily with the documentation?
  - Are the document table of contents and index accurate and complete?
  - Is the design of the document conductive to understanding and quick assimilation of information?
  - Are all error messages displayed for the user described in more detail in the document?
  - If hypertext links are used (in online-documentation), are they accurate and complete?

9.3. Testing for real-time systems

- The time-dependent nature of many real-time applications adds a new and potentially difficult element to the testing: time.
- Next to white-box and black-box testing, the test designers have to consider internal and external event handling, temporal data, transaction deadlines, possible race conditions, and parallelism.

- Comprehensive test case design methods for real-time systems have yet to evolve. However, Pressman presents a four-step strategy (Pressman, 1997):
  - 1. Task testing.
    - The first step in the testing of real-time software is to test each task independently. Task testing uncovers errors in logic and function, but will not uncover timing or behavioural errors.
    - A task here is a separate program that accepts input and produces output. A task may be composed of many modules and can be tested using white-box and black-box methods.

  - 2. Behavioural testing.
    - Using system models the real-time system behaviour is simulated as a consequence of external events.
    - These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built.
    - The test cases can follow black box input class partitioning to categorise events and interrupts for testing purposes.
    - Each event is tested individually. The system is examined to detect event processing errors.
    - Once each class has been tested, events are presented to the system in random order and frequency.

  - 3. Intertask testing.
    - Once errors in individual tasks and external event behaviour have been isolated, testing shifts to time related errors. Asynchronous tasks are tested with different data rates and processing load.
    - Tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
  - 4. System testing.
    - Software and hardware are integrated, and a full range of system tests are conducted in an attempt to uncover errors at the software/hardware interface.

- Real-time systems are deeply related with interrupts. Thus, interrupt code testing is essential to any real-time system testing.
  - Are interrupt priorities correct?
  - Is interrupt execution correct?
  - Do interrupts cause priority inversion (a low priority process blocks a high priority process)?
  - Does a high volume of interrupts at critical times create problems in function or performance?
  - Are transaction deadlines met at critical times with high volume of interrupts?

10. Object-oriented testing

- Current perspectives on object-oriented testing can be characterised on three ways:
  - optimistic: technology and software processes in object-oriented world greatly reduce the need for testing,
  - minimalist: testing remains necessary for object-oriented implementations, but it will be easier and less costly, and
  - expansive: typical features of object-oriented systems require new approaches to attain adequate testing.

- All three perspectives are correct in their own ways:
  - Object-oriented design and component reuse can reduce the amount of newly written code. This in turn reduces tests (optimistic view).
  - The use of object-oriented design does not change any basic testing principles. What does change is the granularity of the units tested (minimalist view).
  - Encapsulation, inheritance, polymorphism, message passing, exceptions etc. need new testing approaches (expansive view).

- Since the optimistic view is more of object-oriented design, and the minimalist view has already been covered, we will concentrate on the expansive view.
- We consider *an object* a uniquely identifiable entity that has *a state* and *a behaviour.*
- The state is defined with a vector of *attributes* each of which may be of a scalar or composite type.

- An attribute may be an object. In such a case the object containing the object attribute is *a composite object.*
- The behaviour of an object is defined with a set of *interfaces.* An interface consists of a set of *public methods* that are accessible outside the object.
- Next to the public methods in interfaces, an object may have a set of *private methods* that are accessible only from inside the object.

- Objects communicate with each other via *messages*. A message has a sender, a receiver, and a service request. The message is sent to one of the object interfaces where the object resolves the request to one of the public methods.
- When the request is resolved, the message is sent back to the sender.
- The object state is reachable only via public methods. Thus, attributes cannot be directly addressed.

- Object-oriented testing has differences to conventional system testing. Four main concerns are often listed in literature:
  - Fault hypothesis. Are some facets of object-oriented software more likely than others to contain faults?
    - What kind of faults are likely?
    - How can revealing test cases be constructed?
    - What effect does the scope of a test have on the chance of revealing a fault?
    - To what extent are fault models specific to applications, programming languages, and development environments?

  - Test case design. What techniques may be used to construct test cases from models, specifications, and implementations?
    - How much testing is sufficient for methods, classes, class clusters, application systems, and components of reusable libraries?
    - To what extent should inherited features be retested?
    - How should abstract and generic classes be tested?
    - What models for understanding collaboration patterns are useful for testing?
    - How can models of the abstract and concrete state of an object, class or system under test be used?
    - How can collection and container classes be tested?

- How can state-dependent behaviour be verified when state control is often distributed over an entire object-oriented application?
- How can clients and servers linked by dynamic binding be tested?
- How can regression tests be formulated and conducted to ensure the integrity of classes?
- How can information in representations and implementations be used to develop or automatically produce the expected results for a test case?

– Test automation issues.
- How should test cases be represented?
- How should test cases be applied? What is the proper role of stubs and drivers?

- How can test results be represented and evaluated?
- When can reports about the state of an untested object be trusted?
- What is an effective strategy for integration testing?
- What role can built-in tests play?
- Should built-in tests be provided by application components, programming languages and/or development tools?

– Test process issues
- When should testing begin?
- Who should perform testing?
- How can testing techniques help in preventing errors?

- How are testing activities integrated into the software process model?
- How can testing facilitate reuse?
- To what extent should reusable test suites be considered a necessary part of a reusable component library?
- What is an effective test and integration strategy given the iterative approach preferred for object-oriented development?

- Source: R. Binder, Testing Object-Oriented Software: a Survey. Software Testing, Verification and Reliability. Vol. 6, pp 125-252 (1996)

10.1. Fault hypothesis

- Two basic issues frame all testing techniques:
  – What is likely to go wrong?
  – To what extent should testing be performed?
- For practical purposes, testing must be based on *fault hypothesis:* an assumption about where faults are likely to be found.

- The object-oriented programming has several unique features: encapsulation, inheritance, and polymorphism. These features are both beneficial and hazardous.
- Thus they are good candidates for fault hypothesis.
- Inheritance hazards:
  – Inheritance is an excellent source for bugs:
    - 1) It weakens encapsulation;
    - 2) fault hazards are introduced which are similar to those associated with global data in conventional languages;

- 3) it can be used as an unusually powerful macro substitution for programmer convenience;
- 4) deep and wide inheritance hierarchies can defy comprehension, leading to errors and reduction of testability;
- 5) semantic mismatches are easily generated since subclass methods have different semantics and require different tests;
- 6) multiple inheritance allows superclass features with the same names, providing the opportunity for misnaming;
- 7) changes to superclasses may trigger bugs to inherited subclasses.

- Encapsulation hazards:
  - When properly used, encapsulation is a safe object-oriented programming feature. In fact biggest errors occur when encapsulation is overridden:
    - 1) direct access to object attributes causes high coupling between objects;
    - 2) lack of use of object interfaces may cause situations where changing any object method will break code in a message sending object.
  - The only negative issue in encapsulation that I can think of is:
    - 3) object encapsulation sometimes makes test case creation difficult by hiding too much.

- Polymorphism hazards:
  - Polymorphism replaces explicit compile-time binding and static type-checking by implicit runtime binding and type-checking. It is a source of many hazards:
    - 1) The deeper is the hierarchy tree and the wider is the tested application, the higher is the likelihood that a dynamically bound method is used incorrectly,
    - 2) the combination of polymorphism and method inheritance make it very difficult to understand the behaviour of subclasses and how they work,

- 3) polymorphism with late binding can easily result in messages being sent to the wrong object since it may be difficult to identify and exercise all such bindings;
- 4) since binding is done at runtime, the type checking responsibility is with the server (instead of the compiler) and the misuse of a message, an interface, or a method becomes extremely simple;
- 5) complex polymorphic relations can serve to confuse the testers and developers, for instance in obscuring the point of origin of a bug;
- 6) subtle naming errors are easily developed and difficult to find.

- While the lists in earlier slides are impressive, they do not exclude *proper* use of the object-oriented tools.
- Rather they show possible error sources that need extra care when generating test cases.
- Fortunately thousands of *design patterns* have been defined to give "recipes" for common object-oriented problem solutions.
- When correctly used, design patterns reduce the number of needed tests.

10.2. Unit testing

- Unit testing is a more general term to testing that we referred to module testing. It describes the smallest entity that is worth a test.
- The smallest distinguishable entity in an object-oriented program is an object. Thus it is a good candidate for unit testing.

- Since objects have behaviour in methods, also a method is a candidate for unit testing.
- Usually an object is chosen as the unit testing entity:
  - most methods are usually very small and trivial to test,
  - object state should be included in unit testing,
  - object interface is a perfect entry for drivers.
- Naturally methods are first tested separately and preliminarily integrated together.

- Similar white-box coverage methods may be used in object-oriented unit testing than in traditional testing.
- When methods are small, each public method can be used as an entry point for white-box coverage test analysis.
- When methods are large, each method can be first used as an entry point and then do integration within the object.
- Also new object-oriented specific coverage criteria must be considered.

- Next to the traditional coverage criteria, we can define at least the following object-oriented coverage:
  - 1) Message coverage. All possible different types of messages that an object can access must be included. Equivalence classes may be used in message parameters to define suitable message-specific test cases.
  - 2) State coverage. Each possible object state must be verified. All states must be accessible and not cause an object deadlock (when the object can no longer change its state).

  - 3) Exception coverage. When object methods support exceptions (as in C++), each possible exception case must be covered. Traditional coverage criteria may be used in the exception code.
  - 4) Interface coverage. All public methods on all interfaces must be tested. Traditional equivalence class partitioning along with boundary analysis may be used in the test case generation.
  - 5) Method coverage. All methods must be reachable and fully white-box tested.

- Black-box unit testing is similar to traditional black-box testing. A driver feeds test cases to a public method in one of the interfaces and receives responses, or the driver creates messages to be sent to the object.
- The main difference between traditional black-box module testing and black-box object oriented testing is that an object has several entry points that are tested separately.

10.3. Object-oriented integration testing

- If we are lucky, objects form a reference hierarchy in the program. In such a case we can use traditional top-down and bottom-up integration methods.
- Unfortunately this is seldom the case. Object-oriented programs are usually built from a set of *subsystems* each of which has an interface and implementation.

- The subsystem interface is similar message passing interface than the object interface.
- The subsystem implementation consist of a set of *interface objects* that implement the subsystem interface, and a set of *local objects* that implement subsystem functionality.
- Thus, object-oriented integration goes in two steps:
  - 1) integrate each subsystem, and
  - 2) integrate subsystems together.

- Subsystem integration may be implemented on two ways: *thread-based* and *use-based.*
- Thread-based integration integrates the set of classes required to respond to one input or event for the subsystem.
- Each thread is integrated and tested individually.
- Regression testing is applied to ensure that no side effects occur.

- Use-based integration begins the construction of the subsystem by testing those low-coupling classes that use very few (if any) of services from other classes.
- Once the low-coupling classes are tested, the next layer of classes that use the services of low-coupling classes is added.
- The procedure continues until the full subsystem is integrated.

- When subsystems are integrated, regression tests are executed to ensure that the integration does not break object behaviour.
- After that subsystems are integrated. Depending on the approach either top-down or bottom-up integration may be used.
- In top-down integration, the main subsystem is first integrated with the next layer subsystems. Then the procedure is repeated with the next layer subsystems until all subsystems are integrated.

- In bottom-up integration the lowest level subsystems that only offer services are integrated first with the next lowest level subsystems. The procedure is continued with the next lowest level until all subsystems are integrated.
- In subsystem integration, extra care must be taken with message and message parameter tests. All possible messages and responses to a subsystem must be tested.

10.4. Later object-oriented testing phases

- When integration testing is finished, we have a complete program.
- At this level the implementation details are no longer visible and thus traditional black-box methods may be used in system and acceptance testing.
- Thus, earlier covered methods may be used in object-oriented testing as well.

11. Conclusion

- Testing is quite a nasty way to validate high quality software:
  - we can't test all inputs,
  - we can't test all possible paths,
  - we often can't even reach all written code,
  - we can't verify that software is free of errors,
  - in fact we can't even find all errors, and
  - as a matter of fact we can't even define for certain that something is an error.

- Then why do we test at all?
  - Because testing is the best that we have got.
- Fortunately a great number of defects, bugs and errors are found in earlier stage inspections and reviews.
- If a project is well managed, most defects have been found and removed when testing begins.
- The managed testing phases and methods help us to find some more defects.

- But no matter how much we test, we can never find that last bug from a nontrivial program.
- When should we then stop testing? Alternatives:
  - when all *planned* tests are executed,
  - when *required* white-box coverage has been reached,
  - when *all* black-box equivalence classes are tested,

  - when required percentage of *estimated* total number of errors has been found,
  - when required percentage of *seeded* errors has been found,
  - when we have reached our *reasonable* testing budget, or
  - when the mean time to failure is greater than a required threshold time.
- Conclusion: Try to minimise needed testing. However, when you have to test, do it carefully and with focus.