Lecture 05: Verification and Validation

Software Verification and Validation (V&V) is the process of ensuring that software being *developed or changed* will satisfy functional and other requirements (validation) and each step in the process of building the software yields the right products (verification) i.e.;

Verification:

"Are we building the product right?"

Verification involves checking that the software should conform to its specification.

Validation:

"Are we building the right product?"

Validation aims to ensure that the software system meets the customer's expectations. It shows that the system does what the user really requires.

V&V Goal and Objectives

The goal of V& V ***is to establish confidence that the software is fit for purpose*** i.e. NOT that it is completely free of defects, rather, it must be good enough for its intended use

V & V activities have two principal objectives:

- Discovery of defects in a system
- Assessment of whether the system is usable in an operational situation

The level of V& V confidence will depend on certain factors;

A. Software function

The level of V&V confidence depends on how critical the software is to an organisation. E.g. mission critical systems require a larger level of confidence than simple game software.

B. User expectations

Users may have low expectations of certain kinds of software and thus will not be surprised when it fails.

C. Marketing environment

Getting a product to market early may be more important than finding defects in the program. E.g. a company with no competitor on the market will not bother for a high V&V confidence level.

The V&V Process

There are two complementary approaches to system analysis and checking within the V&V process:

- Software inspections or peer reviews
  These analyse and check the system representations e.g. requirements document, design diagrams, program source code to find problems. The source code is however never executed and thus cannot be used to check program properties such as performance and reliability. This is also known as static analysis and is a static V&V technique.

- Software Testing
  This is concerned with exercising and observing product behaviour (dynamic analysis). It involves execution of the system implementation with test data as its operational behaviour is observed. It is a dynamic V&V technique.

V& V Planning

V&V is usually a highly cost intensive exercise, therefore, in order to cost effectively get the most out of the V&V process, careful planning is required and should start early in the development process. Each stage of the development process should have a corresponding test plan drawn up which should also identify the balance between static verification and testing.

Test planning is about defining standards for the testing process rather than describing product tests.

Software Inspection

Inspections involve people examining the source representation with the aim of discovering anomalies and defects. They can check conformance with a specification but not conformance with the customer's real requirements.

Inspections do not require execution of code and can thus be done before implementation.

Note: Inspections cannot check non-functional characteristics such as performance, usability, etc.

Inspection Pre-conditions

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal i.e. finding out who makes mistakes.

Roles on the Inspection Team

Normally, inspection involves a meeting although participants can also inspect alone at their desks.

1. Author or owner
   This is the programmer or designer responsible for producing the program or document. They are also responsible for fixing defects discovered during the inspection process.
2. Inspector
   Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
3. Reader
   Presents the code or document at an inspection meeting.
4. Scribe
   Records the results of the inspection meeting.
5. Chairman or moderator
   Manages the process and facilitates the inspection. Reports process results to the Chief moderator.
6. Chief moderator
   Responsible for inspection process improvements, checklist updating, standards development etc.

Inspection Procedure

1. The moderator calls the meeting and the system overview is presented to inspection team.
2. Code and associated documents are distributed to inspection team in advance.
3. The participants prepare for the inspection meeting in advance.
4. At the start of the meeting, the moderator explains the procedures and verifies that everybody has prepared.
5. Readers/Paraphrasers take turns explaining the contents of the document or code, without reading it verbatim. Requiring that the paraphraser not be the author ensures that the paraphrase says what he or she *sees*, not what the author *intended* to say.
6. Inspection takes place and discovered errors are noted.
7. Modifications are made to repair discovered errors.
8. Re-inspection may or may not be required.

Inspection Checklist

This is a checklist of common errors that should be used to drive the inspection. Error checklists are usually programming language dependent and reflect the characteristic errors that are likely to arise in the language. It is the general rule that the weaker the language type checking the larger the checklist.
Examples include: Initialisation, Constant naming, Loop termination, Array bounds, etc.

Specific examples of inspection checks:

Data faults

- Are all program variables initialised before their values are used?
- Have all constants been named?
- Should the upper bound of arrays be equal to the size of the array or Size -1?
- If character strings are used, is a delimiter explicitly assigned?
- Is there any possibility of buffer overflow?

Control faults

- For each conditional statement, is the condition correct?
- Is each loop certain to terminate?
- Are compound statements correctly bracketed?
- In case statements, are all possible cases accounted for?
- If a break is required after each case in case statements, has it been included?

Interface faults

- Do all function and method calls have the correct number of parameters?
- Do formal and actual parameter types match?
- Are the parameters in the right order?
- If components access shared memory, do they have the same model of the shared memory structure?

Exception management faults

- Have all possible error conditions been taken into account?

Input/output faults

- Are all input variables used?
- Are all output variables assigned a value before they are output?
- Can unexpected inputs cause corruption?

Storage management faults

- If a linked structure is modified, have all links been correctly reassigned?
- If dynamic storage is used, has space been allocated correctly?
- Is space explicitly de-allocated after it is no longer required?

Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

There are two different types of testing (also called testing goals) that can be carried out at different stages of the software development process:

- **Validation testing**
  It is intended to show that the software is what the customer wants. A successful test will show that a requirement has been properly implemented.
- **Defect testing**
  In this type of testing, tests are designed to discover system defects. The goal is to find inconsistencies between a program and its specification. A successful defect test is one which reveals the presence of defects in a system.

The overall goal of software testing is to convince the system developers and customers that the software is good enough for operational use i.e. to build confidence in the software.

Defect Testing Vs Debugging
Defect testing and debugging are distinct processes;
Defect testing is concerned with establishing the existence of defects in a program while debugging is concerned with locating and repairing these errors.
These processes are however usually interleaved.

Testing Fundamentals

Objectives of Software testing
- Testing is the process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an *as–yet-undiscovered* error.
- A successful test case is one that uncovers an *as-yet-undiscovered* error.

Who tests the software?
This is a contentious issue where one would be required to select over a trade-off;
- The developer may test the system because he/she understands it, but they will test gently, and will be driven by delivery.
- An independent tester will need to learn about the system first but the upside is that he will attempt to break the system and is driven by quality.

Test completion criteria
How do you know when to finish testing?
Ideally the two most common completion criteria are these:
- Stop when the scheduled time for testing expires.
- Stop when all the test cases execute without detecting errors; i.e. stop when the test cases are unsuccessful.

These are however not usually realised in reality and therefore organisations usually come up with marks or metrics to indicate completion.

Test plan creation

Test plans should be drawn up to guide the testing process. They should not be static as change is usually expected. The recommended structure of a software test plan is usually as follows:

- ✓ The testing process
  This is a description of the major phases of the testing process.
- ✓ Requirements traceability
  Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.
- ✓ Tested items
  The products of the software process that are to be tested should be specified.
- ✓ Testing schedule
  An overall testing schedule and resource allocation for this schedule. This is linked to the more general project development schedule.
- ✓ Test recording procedures
  It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it been carried out correctly.
- ✓ Hardware and software requirements
  This section should set out software tools required and estimated hardware utilisation.
- ✓ Constraints
  Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Effective and Efficient Testing

- Effective Testing
  To test *effectively*, the strategy used should be one that uncovers as many defects as possible. The tester must understand how programmers and designers think, so as to better find defects. Nothing should be left uncovered, and they must be suspicious of everything.

- Efficient Testing
  To test *efficiently*, you must find the largest possible number of defects using the fewest possible tests. It is wasteful to take an excessive amount of time; tester has to be *efficient*.

Testing Policies

Only exhaustive testing can show a program is free from defects. However, since exhaustive testing is impossible, testing policies have been developed to define the approach to be used in selecting system tests:

- All functions accessed through menus should be tested;
- Combinations of functions accessed through the same menu should be tested;
- Where user input is required, all functions must be tested with correct and incorrect input.

Testing Principles

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 customer requirements.** Since the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- **Tests should be planned long before testing begins.** Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified.
- **The Pareto principle applies to software testing.** The Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
- **Testing should begin "in the small" and progress toward testing "in the large".** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
- **Exhaustive testing is not possible.** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
- **To be most effective, testing should be conducted by an independent third party. '***Most effective',* means testing that has the highest probability of finding errors. Thus the software engineer who created the system is not the best person to conduct all tests for the software.

Test Case Design

Considering the objectives of testing, tests must be designed, that have the *highest likelihood of finding the most errors with a minimum amount of time and effort.*
Test-case design is thus an attempt to try to make tests as complete as possible.

The goal of test case design is to create a set of tests that are effective in validation and defect testing.

The Structure of a Test-case

A. Identification and classification:

- Each test case should have a number, and may also be given a descriptive title.
- The system, subsystem or module being tested should also be clearly indicated.
- The importance of the test case should be indicated.

B. Instructions:

- Tell the tester exactly what to do.
- The tester should not normally have to refer to any documentation in order to execute the instructions.

C. Expected result:

- Tells the tester what the system should do in response to the instructions.
- The tester reports a failure if the expected result is not encountered.

D. Cleanup (when needed):

- Tells the tester how to make the system go ‘back to normal’ or shut down after the test.

Test-case Design Techniques

Deriving from the different approaches to test case design, software is tested from two different perspectives:

1. Tests can be conducted to ensure that the internal operations are performed according to specifications and all internal components have been adequately exercised -“white box” test case design techniques.
2. With the knowledge of the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function (software requirements) - “black box” test case design techniques.

Black-box testing
This is a software testing technique in which the internal workings of the item being tested are not known by the tester. *E.g.*, in a black box test on software design the tester only knows the inputs and what the expected outcomes should be and not how the program arrives at those outputs. Black-box test design treats the system as a "black-box", so it doesn’t explicitly use knowledge of the internal structure. Black-box test design is usually described as focusing on testing functional requirements.

The tester does not ever examine the programming code and does not need any further knowledge of the program other than its specifications.

Examples of black-box testing methodologies include: *Equivalence partitioning, Boundary-value analysis, Cause-effect graphing and Error guessing.*

White-box testing
Also called ‘glass-box’ or ‘structural’ testing, this is a software test case design technique whereby explicit knowledge of the internal workings of the item being tested is used to select the test data i.e. allows one to peek inside the "box".

Unlike black box testing, white box testing uses specific knowledge of programming code to examine outputs. The test is accurate only if the tester knows what the program is supposed to do. He or she can then see if the program diverges from its intended goal.

The testers therefore have access to the system design and they can:

- Examine the design documents
- View the code
- Observe at run time the steps taken by algorithms and their internal data

Individual programmers often informally employ glass-box testing to verify their own code.

Note: White box testing does not account for errors caused by omission, and all visible code must also be readable.

Using white-box testing methods, the software engineer can derive test cases that

1. Guarantee that all independent paths within a module have been exercised at least once,
2. Exercise all logical decisions on their true and false sides,
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise internal data structures to ensure their validity.

Examples of white-box testing methodologies include: *Statement coverage, Decision coverage, Condition coverage, Decision-condition coverage, Multiple-condition coverage.*

The Testing Process and Stages

Testing usually begins with *functional* (black-box) tests, is supplemented by *structural* (white-box) tests, and progresses from the unit level toward the system level with one or more integration steps.
The testing process is generally phased (levels);

1. **Component testing**
   A.K.A unit testing, it is the process of testing individual components in the system e.g. classes, functions, packages. The goal is to expose the faults in the components. This testing is usually done by the developers of the system and tests are usually derived from the developer's experience.

2. **System testing**
   This testing phase involves integrating two or more components that implement system functions and then testing the integrated system. It is the responsibility of the independent testing team and tests are usually based on a system specification.

   There are two distinct system testing phases:
   - Integration testing
     The test team have access to the system source code and when a problem is discovered the integration team finds the problem source and identifies the components to debug.

     This testing therefore involves building a system from its components and testing it for problems that arise from component interactions. Integration may take two forms:
     - ✓ Top-down integration develops the skeleton of the system and populates it with components.
     - ✓ Bottom-up integration integrates the infrastructure components then adds functional components.

   - Release testing
     A version of the system that could be released to the customer is tested, validating that it meets the requirements and ensuring that the system is dependable. When customers are involved in release testing it is referred to as *acceptance testing.*

Other types of testing include:

- Regression testing
  This is a re-testing procedure done to detect problems caused by the adverse effects of program change.
- Alpha testing
  This is the actual end-user testing performed within the development environment i.e. *alpha test* is conducted at the developer's site by a customer.
- Beta testing
  The end-user testing performed within the user environment prior to general release i.e. *beta test* is conducted at one or more customer sites by the end-user of the software. Here the developer is generally not present.

Testing Guidelines

Testing guidelines are hints for the testing team to help them choose tests that will reveal defects in the system. These include:

- Choose inputs that force the system to generate all error messages;
- Design inputs that cause buffers to overflow;
- Repeat the same input or input series several times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small.

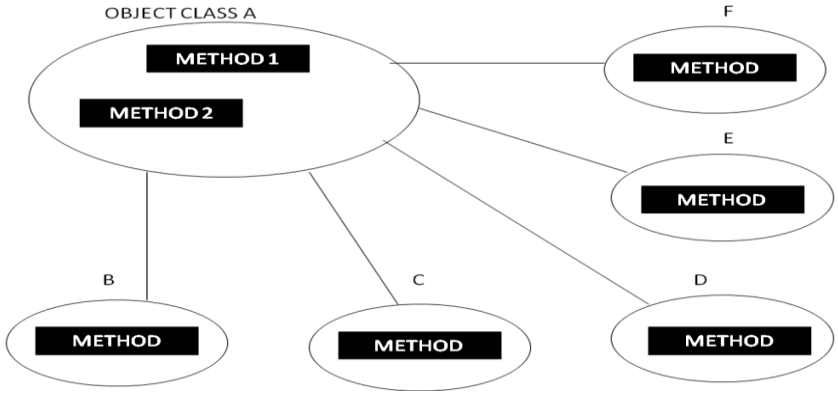
Object Oriented Testing

OO testing is strategically similar to the testing of conventional systems, but it is tactically different. Once code has been generated, OO testing begins "in the small" with class testing.

Unit Testing in the OO Context

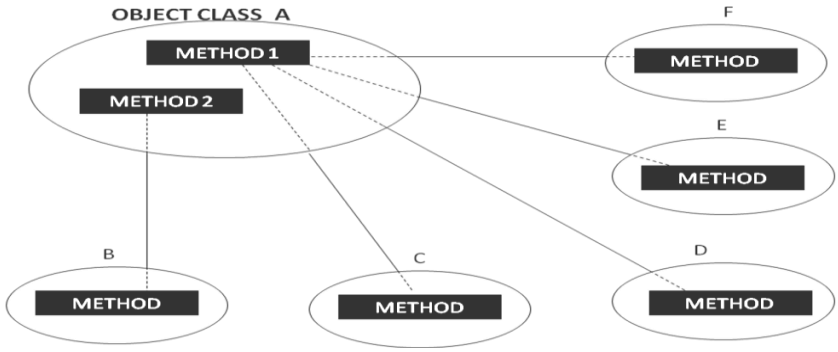
OO software changes the concept of the unit. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as methods or services) that manipulate these data.
Unit testing of OO software is generally performed in the same manner as unit testing of non OO software. The only difference is what the definition of a unit in OO software design and testing is. There are currently two main stream approaches: METHOD & CLASS.

UNIT CLASS



In the diagram above the Object Class "A" has two methods and a dependency association with all other classes as shown. If a change was made to the unit "Object Class A" (METHOD 2), in theory all test cases involving the unit should be rerun, at all levels.

UNIT METHOD



In the diagram above using both METHOD 1 and METHOD 2 of Class A as units, when a change is made to METHOD 2, only test cases involving METHOD 2 need to be rerun. In this case, this would only involve one method from Object Class B.

Note: In order scale up to larger systems, whether a method or a class is used as a unit, there is definite requirement for some form of automated tool that can quickly tell a tester the effect of making changes(what test cases should be rerun (regression testing)).

Integration Testing in the OO Context

Integration testing is the most complicated part of OO testing and it is done in a bottom up approach and is based on the behavior of the software versus the structure. Methods and classes are unit tested and then composed with other classes and their methods to perform integration testing.

There are two different strategies for integration testing of OO systems.

1. **Thread-based testing**
   Integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur.
2. **Use-based testing**
   This approach begins the construction of the system by testing those classes (called independent classes) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

Note: Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations.

Validation Testing in an OO Context

At the validation or system level, the details of class connections disappear. Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable output from the system. To assist in the derivation of validation tests, the tester should draw upon the use-cases that are part of the analysis model. The use-case provides a scenario that has a high likelihood of uncovered errors in user interaction requirements.

Conventional black-box testing methods can be used to drive validations tests.

Inspection vs Testing

- Both testing and inspection rely on different aspects of human intelligence.
- Testing can find defects whose consequences are obvious but which are buried in complex code.
- Inspecting can find defects that relate to maintainability or efficiency.
- The chances of mistakes are reduced if both activities are performed.

Testing or inspecting, which comes first?

It is important to inspect software *before* extensively testing it.

- The reason for this is that inspecting allows you to quickly get rid of many defects.
- If you test first, and inspectors recommend that redesign is needed, the testing work has been wasted.

References

Sommerville, I., *Software Engineering*, 8th ed., Pearson, 2006.
Myers, Glenford J., *The Art of Software Testing*, 2nd ed., Wiley, 2004
Pressman, Roger S., *Software Engineering-A Practitioner's Approach*, 5th ed., McGraw-Hill, 2001.