

CS1Ah Lecture Note 26

Validation and Verification

This note has four main aims:

1. To introduce the notions of *validation* and *verification* and their role in developing software.
2. To introduce *testing* as a way of doing verification.
3. To provide an introduction to *defect testing*.
4. To introduce some methods for exploring the *adequacy* of a test set.

Validation and verification are two processes that should go on throughout the process of developing and maintaining software. Their purpose is to try to ensure that the system meets the needs of the those who have an interest in the system (e.g. users, purchasers, people affected by the use of the system). Recently the word *stakeholder* has come to be used as a shorthand for those with an interest in the system. Working definitions of the terms are:

Validation: This is the process of checking that our idea of the requirements matches the *real* requirements (i.e. what the stakeholders want the system to do). This is usually referred to as checking we are *building the right thing*. The difficulty with validation is in checking that our model of the system (which will be embodied in the implementation) matches up with the stakeholders expectations. The problem with this is that stakeholders may have inconsistent and conflicting ideas of what the system should do and their ideas will change through time. In particular, they will change as soon as we show them a preliminary implementation or a mock-up of a design.

Verification: This is the process of checking that the system we are designing and implementing is consistent with the requirements we have identified. This is usually referred to as checking we are *building the thing right*. This process is internal to the design process and implementation process and involves checking the developing system is consistent with our idea of the requirements.

Validation and verification go on throughout the development and maintenance process — each design or implementation decision should be validated and verified — of course this is not often the case — but strategies have been developed which help to ensure a system is reasonably validated and verified before it becomes operational.

Validation and verification activities fall into two broad categories:

Dynamic: These activities focus on the execution behaviour of the (partially developed) system. Either by simulating or by executing components of the code in suitable test harnesses the aim is to use the behaviour of the system in its validation or verification. Often simulations are used early in the development process to validate design decisions while later in the process testing attempts to detect defects in the code for the system.

Static: These activities focus on the text of the system and aim to discover defects by subjecting it to scrutiny. Static validation and verification can range from the process of code reviews and walkthroughs where a group including the author of the code examine it in detail to the mathematical analysis of the code in an attempt to show that mathematically expressed properties hold of the system. In later lectures we consider some mathematical techniques for proving that a program has a particular property.

This note concentrates on dynamic methods. In particular, we discuss testing as a means to carry out validation and verification. Testing is a major part of most software development projects consuming a significant proportion of the project resources.

26.1 Testing

The aim of testing is to predict some aspect of the overall behaviour of the system on the basis of the behaviour of the system in some particular cases. Because most real systems have a very large number of states and possible inputs we only observe a very small proportion of the potential behaviours of the system under test. Thus testing has its limits but it does appear to be useful in practice. In particular:

- The effectiveness of testing is based on assumptions that the behaviour of the program under test does not change significantly for small changes in input (so we don't need to look at lots of similar test cases).
- Testing is a good method for discovering problems with a system. If a system passes all the tests we have prepared for it we cannot necessarily confidently predict that the system has no problems.
- It may not be possible adequately to check some property of the system exclusively by testing.

Observing the system executing on one test usually provides very little information. In carrying out testing we are concerned with constructing *test sets* that are concerned with uncovering some particular aspect of the system.

Why test?

There are many different reasons why we might want to test a system. There are two main sub-divisions:

Functional testing: here we are looking at the system in an attempt to uncover *defects* in the code that cause it to fail to provide the expected functionality. This is often called *defect testing*. A successful test in this kind of testing is one which uncovers some kind of error.

Statistical testing: here we are trying to make some statistical estimate of an attribute of the system e.g. the typical running time of the system, the reliability of the system (i.e. the mean time to failure for the system). We may also try to estimate some attributes that are more difficult to quantify, for example, the usability of the system.

Test what?

Testing can be grouped into three broad categories depending on the entity and characteristics being tested:

Unit: this type of testing is oriented to discovering defects in the individual methods and classes making up the system. Because it involves small components deeply embedded within the system this is mainly a verification concern. Good practice suggests that unit tests should be constructed by someone other than the author of the particular unit though in practice the developer also develops the test.

Integration: These are tests at the module, sub-system and system level. As we move to higher levels of integration the purpose of testing moves from predominantly verification to validation as we approach the full system. In object-oriented systems we use groups of objects to achieve particular sub-systems of the whole system. These groups should be tested to see they work together properly as we build the system.

Acceptance: This level of testing is entirely concerned with validating the design — acceptance testing should not be deferred to the time when the final system is delivered. Prototypes, simulations etc. can be subject to acceptance testing early in the development process. This outcome of early acceptance testing is a requirement that the delivered system conform to the prototype or simulation. Such requirements require verification testing as the design proceeds.

Test planning

The process of planning system testing proceeds as the design is developed:

Acceptance test plan: this should be generated as the requirements are determined. This concentrates on ensuring the acceptance test covers all the system requirements. The conceptual design of the system should also play a rôle in deciding on the acceptance tests.

System integration test plan: this is generated from the technical design specification and some high level characterisation of the system design.

Sub-system integration test plan: this utilises some low-level aspects of the system design and the detailed implementation design.

Module and unit tests: these are developed during detailed design.

The outcome of such planning is the set of tests which will be applied to the system as it is developed and integrated.

26.2 Test Strategies

There two broad types of test which we can apply to systems. These are:

Functional testing: here we are interested in checking the behaviour of an incomplete system. In non-object-oriented systems the kinds of incompleteness that usually arise are the absence of low-level modules in top-down development or the absence of higher level modules in bottom-up development. In both cases the usual procedure is for the tester to write code to simulate the low-level code (these are usually called *stubs*) or the high level code (these are usually called *drivers*). In the case of OO systems this distinction is less useful. Often code develops by extending classes with new methods. In this case the test set should be structured to test each new layer of methods as they are added to the classes of the system.

Stress testing: all implementations of systems have resource limits which complicate the behaviour of the system and may render it inoperative if they are exceeded and the programmer takes no account of the possibility of such failure. Typical limitations are: storage, compute time, communication capacity, database retrieval time. Stress testing should be designed to drive the system to the point where these limits are reached. Such tests often reveal the omission of features in the implementation designed to deal with such limitations, they also often turn up potential defect which could result in catastrophic failure (e.g. large scale loss of data or service).

26.3 Test set criteria

In designing test sets we can choose different criteria of adequacy depending on what information we decide to look at in making the judgement. The three we mention here are functional, structural and residual defect criteria.

Functional criteria

This is also called *black box* testing — given a system and its intended function we should test it provides that function without consideration of the code used in the implementation. This means we should inspect the requirement and try to identify a collection of test cases that cover all the distinctively different kinds of behaviour the system is required to exhibit and choose tests to exercise the program in a way that will cover all theses behaviours.

For example, if we are testing the student record system we considered earlier in the course we should at least have tests that add, delete and search for a particular student.

Structural criteria

Structural testing is also called *glass-box* testing because in this process the tester is permitted to look at the code rather than just its specification. In this form of testing the emphasis is on ensuring that all the code in a system has been exercised by the test set, and, if possible, all possible paths through the system have been followed. The tester uses the code of the system to generate test cases to force the code under test through a particular sequence and may use tools like test coverage monitors to check the level of coverage.

This form of testing complements and refines black-box testing since seeing the structure of the code allows the tester to refine the black-box test cases to cover all the code. However, black-box testing should be undertaken first since it allows the tester to be independent of the implementation choices made in the module.

Residual defect criteria

This is a technique which is commonly used to explore the efficacy of the test set for a system. The system is seeded with a number of “common” errors and the test set is then run on that version of the system. The number of detected defects is then used to provide evidence of the efficacy of the tests and hence provide evidence of how many errors could remain in the real system. The basis of this method is rather dubious but it does provide some indication of the “efficiency” of a test suite in detecting errors.

Defect Testing

In designing tests to uncover defects the following broad guidance should be followed:

- testing should be aimed at capabilities rather than components. The decision to test some unit in a particular should be motivated by its interest in providing some capability to the user rather than it being of some interest to the implementation.
- If the system is being revised or maintained concentrate on the existing functionality (which the users expect to remain unchanged) since maintaining its performance is more important than ensuring flawless performance of the new components.
- Concentrate on typical data rather than extremes and boundary conditions — in use these boundary conditions almost never occur whereas performance on typical data should be as good as possible.

26.4 Debugging

Once a defect has been revealed the process of *debugging* attempts to:

locate, the defect in the design or implementation. This is far from a trivial problem — it may be possible to locate the defect in different parts of the design. For example the defect may relate to validation if a real requirement is not met and the implementor has correctly implemented an erroneous requirement or it may relate to verification if the requirement is correct but the implementor has failed to correctly implement the requirement.

re-design, that part of the system to eliminate the defect. This process may require major reconsideration of the overall design and, in practice, can often result in the introduction of new defects which were not present in the original design. The re-design process is crucial to the effectiveness of testing — if it is not undertaken with full consideration of the intended behaviour of the component it will often lead to the introduction of serious defects.

re-implement, the revised design.

re-test, the component, *with all tests for that component*. This is an important part of debugging, the component in which the fault has been eliminated must be completely tested in order that defects which might have been introduced can be revealed. Of course this is not a foolproof approach because the new defects may not be revealed by old tests — but at least we know the new component passes more tests than the old one (more — because we have redesigned the old system to eliminate a defect discovered by some test).

26.5 Testing in practice

A wide range of testing tools are available. Structural testing is usually supported by some form of *profiling* the execution of a program on test input. This allows the tester to assess the adequacy of the test set to see if the required coverage criterion is met by the test set. These tools give an idea of when statements have been executed in the course of testing.

Debugging is usually supported by a debugging tool that lets the programmer control the execution of the program on a failed test. Typical features include the ability to look and alter the state of the system, to execute until some point in the program is reached and to check a property is maintained by the system. For Java the debugger is called **jdb** and it provides a typical range of facilities. We'll demonstrate it in CS1Bh.

Often programmers are encouraged to develop and test very frequently. To help this it is useful to give the programmer an environment in which to manage and develop test sets as the program is developed. A typical example of this is **JUnit**¹. This offers facilities to manage and apply a test set as the developer codes the system.

*Lecture note by Stuart Anderson.
David Aspinall, 29th November 2002.*

¹See: <http://www.junit.org>