Software Unit Testing

Rodney Parkin, IV&V Australia

This paper is an overview of software unit testing. It defines unit testing, and discusses many of the issues which must be addressed when planning for unit testing. It also makes suggestions for appropriate levels of formality and thoroughness of unit testing on typical development projects.

What is "Unit Testing"?

The software literature (notably the military standards) define a *unit* along the lines of *the smallest collection of code which can be [usefully] tested*. Typically this would be a source file, a package (as in Ada), or a non-trivial object class. A hardware development analog might be a PC board.

Unit Testing is just one of the levels of testing which go together to make the "big picture" of testing a system. It complements integration and system level testing. It should also complement (rather than compete with) code reviews and walkthroughs.

Unit testing is generally seen as a "white box" test class. That is, it is biased to looking at and evaluating the code *as implemented*, rather than evaluating conformance to some set of *requirements*.

Why is it important?

For any system of more than trivial complexity, it is highly inefficient and ineffective to test the system solely as a "big black box". Any attempt to do so quickly gets lost in a mire of assumptions and potential interactions. The only viable approach is to perform a hierarchy of tests, with higher level tests assuming "reasonable and consistent behaviour" by the lower level components, and separate lower level tests to demonstrate these assumptions.

It would be infeasible to test a space shuttle as a system if you had to simultaneously question the design of every electrical component. It is similarly infeasible to test a large software system as a whole if you have to simultaneously question whether every line of code, every "if statement", was correctly written.

Boris Beizer has defined a progression of levels of sophistication in software testing. At the lowest level, testing is considered no different to debugging. At the higher levels, testing becomes a mindset which aims to maximise the system reliability. His approach stresses that you should

"test" in the way which returns the greatest reliability improvement for resources spent rather than mindlessly performing some "theoretically neat" collection of tests.

Experience has shown that unit-level testing (and reviewing) is very cost effective. It provides a much greater reliability improvement for resources expended than system level testing. In particular, it tends to reveal bugs which are otherwise insidious and are often catastrophic – like the strange system crashes that occur in the field when something unusual happens.

What should it cover?

Just as a system needs to be *designed* before it can be effectively *implemented*, so too must the system test strategy be designed before it is implemented. At the same time as the system concepts are emerging and an architecture is being worked out, a "test strategy" must also be developed.

The test strategy should identify the totality of testing which will be applied to the system – what types of testing will be performed, and how they will contribute to the overall quality and reliability of the product. A good test strategy will clearly scope each class of test and assign responsibility for it. Typically an organisation will have some standard conventions to follow, but each project must identify aspects of the system which are critical or problematical, and clearly identify the how these will be tested and by whom. Ultimately the Project Plan (or some form of Master Test Plan) for each project will define what needs to be covered by unit testing on that project. This type of information works well presented in a checklist.

Usually unit testing is primarily focused on the implementation – *Does the code implement what the designer intended?* For each conditional statement, is the condition correct? Do all the special cases work correctly? Are error cases correctly detected?
However many systems have some high-level requirements which are difficult to adequately test at a system level, and it is common to identify

these as additional test obligations at the unit-test level. An example is detailed signal processing algorithms. These may be fully specified at the system functional requirements level, but it may be most efficient to test the details of the processing at the unit-test level, with system-level testing being confined to testing the gross flow of data through the system.

Who should do it?

Because unit testing is primarily focussed on the implementation, and requires an understanding of the design intent, it is much more efficiently done by the designers rather than by independent testers.

There are some theoretical arguments that it is better for testing to be done independently. However, in this case, the lost efficiency in having an independent person understand the code and understand the design issues strongly outweighs any advantages. Beizer's principal of applying available resources in the most efficient way applies. The benefits to be gained by independence are achieved more easily in a review or walkthrough forum.

What level of formality is required?

When considering the level of formality required for unit testing, the sort of questions which arise are: Do unit tests need a "pre-approved" test protocol, or is it sufficient for them to be worked out "as you go"? Is a formal report required? Do QA need to be involved? Are all results reviewed?

The level of formality required for unit testing depends on your "customer" needs. Where development is being done under a contract with an external customer, or there are regulatory requirements to be met, these my impose specific standards on the project.

Where there are no specific requirements imposed on the project, it becomes essentially a tradeoff between project cost and risk. In fact, the project may choose to keep the level of testing in some areas quite informal, while other are more formal.

These questions should be answered as part of the test planning, and need to be documented in the project test plan. In most cases there is little advantage in requiring any more formality than is required to ensure that adequate attention is being applied to the task. This may need nothing more

than regular liaison and one-on-one review with the tester's team leader.

What type of documentation is required?

Like the required level of formality, the appropriate level of documentation for unit testing varies from project to project, and even within a project. There may be minimum standards imposed by outside agencies, but generally there are not.

The minimum requirements for the documentation are:
- It must be *reviewable*. That is, the records must be sufficient for others to review the adequacy of the testing.
- It must be sufficient for the tests to be *repeatable*. This is important for regression testing - unless you are sure you can repeat a test, you can never be sure if you have fixed the cause of a test failure. Repeatability is also important for analysing failures – both failures during the initial testing, and subsequent failures. Knowing exactly what was and was not tested, and exactly what passed and what failed during testing is an invaluable aid in isolating difficult-to-reproduce field failures. Repeatability not only implies the need to record in reasonable detail how the test is run and what data is used, but also implies identification of the version of code under test.
- The records must be *archivable*. That is, they must be sufficiently well kept and identified that they can be found if required, at a later time (perhaps years later when analysing a field failure).

For many organisations, separate unit test documents are not produced. Typically unit testing will be recorded in controlled lab-books, or collected into project journals.

One approach which works well for software unit testing is to use a source code listing with hand annotations for the recording of tests. Test cases and data are identified on the listing, with markups showing which sections of code are covered by which tests. Typically this listing will be attached to a review sheet and a checklist of unit testing requirements, and filed with the project records.

The documentation method chosen may vary depending on the criticality, complexity, or risk associated with the unit. For example, in a security-critical system, one or more units

associated with the secure interface may be required to have formally documented unit tests, while the (non-security critical) bulk of the system is much less formally documented. These decisions need to be made early as part of the initial project test planning and appropriately recorded.

How "thorough" does it need to be?

In general terms, unit testing should provide confidence that a unit does not have unpredictable or inconsistent behaviour, and that it conforms to all the "design assumptions" that have been made about it. If this is achieved, then higher-level testing can concentrate on macroscopic properties of the system, rather than having to iterate over numerous possibilities for interaction at the lowest levels. In choosing tests, the tester should consider whether it behaves in the way the design assumes, whether it does this over the full range of operational possibilities, and whether there are any "special cases" in its behaviour which are not visible at a higher level. For each line of code, the tester should ask "does it achieve what it was put here to do"?

Because unit testing is primarily implementation driven, its thoroughness is usually measured by *code coverage*. Tools are available which will evaluate code coverage while tests are being run, but generally someone familiar with the code, while focussed on a particular unit, will find it quite easy to determine the coverage of a particular set of tests. Various "measures" of coverage can be defined, such as "statement coverage" (each statement executed at least once), "decision coverage" (each conditional statement executed at least once each way), and so on.

Like documentation, the level of thoroughness required for unit testing may depend on the criticality, complexity, or risk associated with the unit. For example, safety or security-critical units may be subjected to much more extensive unit testing than non-critical screen-formatting code. Some projects use metrics such as McCabe Cyclomatic Complexity to pre-determine the appropriate level – units with a high complexity are required to have a greater degree of testing. Again a policy on test rigour needs to be determined as part of the early project test planning.

For typical projects, the usual standard is to aim for "decision coverage". That is, unit testing must demonstrate correct operation over a range

of cases which require every statement to be executed at least once, and every conditional statement to go each way. In addition, all "boundary cases" must be exercised. In actual practice, 100% coverage can be surprisingly difficult to achieve for well-written code. This is because there will be code to protect against "should not occur" scenarios, which can be very awkward to exercise. A code coverage standard may concede coverage of these cases so long as they are adequately desk-reviewed.

What "test environment" should be used?

As a general rule of thumb "the rest of the system is the best test harness" for unit testing. Performing unit tests in a system environment maximises your likelihood of identifying problems. On the other hand, the tester should not allow this "rule" to limit or hinder their testing. They should use the rest of the system to generate and analyse test scenarios, but should not feel constrained from intruding into the system with debuggers, special test code, or other aids.

Some people feel that for testing to be valid, it must be performed on exactly the code to be delivered, running exactly in its final environment. Although this is appropriate for final acceptance testing at the system level, it can actually be counter-productive at the lower levels. At the unit test level it is far preferable to "put in some debug statements" to help perform a particular test, than to avoid the test altogether in a mistaken attempt to ensure fidelity.

It is often easy to make the system an almost ideal test harness. For example, removing restrictions on selectable system parameters when in a "system test mode" may make it trivial to force otherwise difficult "should not occur" special cases. Providing a capability to inject arbitrary byte sequence for internal messages may be trivial to implement buy extremely useful for testing. When considered early in the design process, these sorts of capabilities are often trivial to provide.

Conclusions

Software unit testing is an integral part of an efficient and effective strategy for testing systems. It is best performed by the designer of the code under test.

The appropriate level of formality and thoroughness of the testing will vary from project

to project, and even within a project depending on the criticality, complexity, and risk associated with the unit. The policy in this regard should be decided early in test planning, and documented, usually in the Project Plan or separate Master Test Plan.

In most cases it is acceptable to adopt an approach which requires little documentation overhead. However there are some basic requirements which should always be met. In particular it must be reviewable, repeatable, and archivable. Commonly, unit testing will be recorded in labbooks, or in hand-written notes on code listings stored in the project journal, with guidance provided by a checklist that identifies the required unit testing activities.

Some issues which should be considered when evaluating a unit testing strategy are:

- Has a policy with regards formality, documentation, and coverage been determined early enough in the project?
- Does it relate to other levels of testing to give an efficient and effective overall strategy?
- Have the needs of units which are particularly critical, complex, or risky been considered?
- Will the documentation be reviewable, repeatable, and archivable?

Questions which should be considered when evaluating unit testing for adequacy include:

- Have all statements been exercised by at least one test?
- Has each conditional statement been exercised at least once each way by the tests?
- Have all boundary cases been exercised?
- Were any design assumptions made about the operation of this unit? Have the tests demonstrated these assumptions?
- Have the tests exercised the unit over the full range of operational conditions it is expected to address?