Developers and Testers – Who Should Do What?

By Rodney Parkin, IV&V Australia

Contact details *IV&V Australia Pty Ltd*
Suite 3A, 10-12 Clarke Street, Crows Nest, NSW, 2065
Phone +61 2 9957-6577
Email rodney.parkiin@ivvaust.com.au; URL http://www.ivvaust.com.au

Abstract *On many projects the division of testing responsibility between developers and independent testers is not well understood. This can lead to gaps in test coverage, as well as strained relationships between the two groups. In our consulting work, we find that this conflict is almost always caused by a lack of a clear understanding of the respective roles and responsibilities of these two groups.*

This paper discusses some of the causes of the conflict, presents a simple framework for defining the different team roles, and provides some practical methods for defining "who should do what".

Introduction

In many development organisations there seems to be an almost inevitable conflict between the developers and the independent testers.

We often see this conflict in our consulting work – particularly in organisations with less mature development processes. Typically we find that both groups are diligently trying to "do their bit", but see the other group as letting them down.

We have come to recognise that the root cause of this conflict is commonly a lack of understanding of the test responsibilities of the two groups. This lack of understanding can lead to a focus of effort that is not only inefficient for both groups, but also encourages unnecessary gaps and overlaps in test coverage.

Although there is no universally *right* test strategy – this depends on the project and the available team – we have identified a basic division of responsibility which is simple to understand but surprisingly effective. Even if you don't plan anything else, following this basic strategy as a starting point will provide a significant improvement in test effectiveness.

The Developer/Tester Conflict

Very commonly we hear testers say *"I just can't run my tests – the software is not stable enough!"* In these cases, the testers spend most of their time after software hand-over in identifying and documenting basic robustness issues.

The testers may have invested a lot of time and effort in planning and preparation, but they are not able to get far enough through their test scripts to even determine a pass/fail status. The software is just not stable enough to support functional and system testing. Much of the effort that they spent in preparation is wasted, and they feel that they are relegated to just "debugging" someone else's software.

When operating in this mode, the testers often feel undermined by the developers, and personal conflicts flourish.

At the same time, the developers also see a problem. A common complaint is *"the testers don't have a deep enough understanding of the implementation of the system to test it properly"*. To them, the problem reports seem arbitrary and unsystematic, and they feel that the testing does not exercise the "real" issues with the software.

The developers know that testing is necessary, but they see the testers as ineffective. This leaves the developers feeling let down by them. They say that the testers are not doing their jobs properly, and as a result they are forced to spend time doing more testing – "someone else's job."

The net result is that both teams feel they are not able to do their own job properly. Even with the best planning and preparation, testers spend most of their time trying to make sense of system crashes and other unpredictable failures – an activity they have neither the skills nor interest to do well. This leaves them less time to do their "own" testing. At the same time, developers feel that they are spending their valuable development time doing testing jobs which they shouldn't have to do.

We end up with a situation where both teams are performing "test" activities which they are neither interested in nor good at. The result is inadequate test coverage and interpersonal conflict.

How to Get the Most out of Testing

There are a few simple rules which should be considered when deciding on a basic test strategy. These rules are:

- Assign the test tasks to the people with the most aptitude for them;
- Order the test tasks for maximum efficiency; and
- Concentrate on the most effective test types.

As we explore each of these rules, you will see that they all lead to similar conclusions about a basic test strategy.

Assign the test types by aptitude

Not long after the requirements have been defined, developers start to become more and more focussed on implementation issues. They tend to use the design documentation as their reference point for implementation, rather then the requirement documentation. As a result, their interpretation of the requirements becomes influenced by the design (which may have diverged from the requirements over time).

By necessity, developers become very familiar with the system design and implementation details (because they are the implementors). Therefore, they are most *efficient* at designing and performing tests based on the design or code structure (ie white-box tests). However, developers testing their own code (and even developers testing each other's code) are likely to miss functional errors and omissions (because they are likely to suffer exactly the same oversights that cause these defects in the first place).

On the other hand, Testers traditionally focus is on the system requirements. They are responsible for ensuring that the developed product meets the user's needs. As such, they can become quite *effective* at functional, black-box testing.

For testers to acquire the same knowledge of the system implementation as the developers, they would have to spend a considerable of time learning about these issues as an additional task to their usual responsibilities.

Based on this observation, it is more efficient for developers to concentrate on structural (ie white-box) tests and testers to concentrate on functional (ie black-box) tests.

Order the testing for maximum efficiency

There is an old development maxim which goes *"Make it work, [then] Make it right, [then] Make it fast"*. This recognises the fact that it is ineffective to worry about code optimisation until the code at least performs the correct functions, and it is inefficient to worry about detailed functional correctness if the system is fundamentally unreliable.

This same principle can be applied to testing. The earliest levels of testing should focus on *"making the system work"* – that is, on identifying (and fixing) reliability issues and thus ensuring basic robustness. Only when these issues have been identified and addressed should the focus shift to *"making it right"* – that is, to detailed verification of functional correctness.

Following this rule leads us to conclude that any testing performed by the developers prior to handing over the software to the testers should be primarily focussed on establishing reliability. Testing for functional correctness by the independent testers should not start until after this has been done.

To be effective, the handover from developer to test needs to be a controlled process with clear exit and entry criteria. It is important that you do not define successful completion of coding merely as the act of handing software over to the independent testers. If you do this, you can inadvertently reward the developers for handing over unreliable/untested software.

Concentrate on the most effective test types

An effective test strategy incorporates the use of a variety of different types of testing. For example: testing can occur at the unit, integration, system and acceptance levels; test strategies can incorporate both white-box and black-box techniques; and testing can be done by developers or by independent testers.

Of all the possible test types, which ones give the best return for resources expended? In the final analysis, the answer comes down to which attributes of a system most affect customer satisfaction. These are typically Reliability and Functionality.

Our experience is that white-box/structural testing done by developers at the unit and integration level is most effective at finding Reliability issues and that black-box/functional testing done by independent testers at the system level is most effective at identifying Functionality problems.

The conclusion from this rule is again that developers should concentrate their testing on white-box testing at the unit and integration level; and testers should concentrate on black-box/functional testing at the system level.

As a second priority, to further strengthen the test strategy, other test types may need to be considered. However our experience is that many of these tests can be more effectively tackled by static review rather than by dynamic test.

An Entry-Level Test Strategy

An effective entry-level test strategy should be designed so that the testing activities provide the best return on the effort expended. The strategy should also provide clear guidance on who should do what testing tasks during the development lifecycle.

At a basic level, the strategy does not have to be extensive nor overly formal to be effective.

Reasonable efficiency and effectiveness can be obtained by following a few simple rules with respect to the Developer and Independent Test Strategies, an associated Review Strategy, and by appropriately managing Software Handover.

Developer Testing

As a priority activity, the developers should perform unit and/or integration testing of their code prior to handover to the test group. This testing should be directed as follows:

1. The primary test objective should be to show *robustness* at the unit and integration test levels.

   That is, it should show that the system behaves in a consistent and predictable manner irrespective of all the special cases and exceptional conditions which can arise at the code level. Verifying functional requirements should be of lower priority.

2. The testing should be structural and should focus on special/extreme values and on exception handling.

3. The test plan should aim to maximise code-based metrics such as statement or decision coverage.

Independent Testing

Also as a priority activity, the independent testers should perform functional and system testing after handover. This testing should be directed as follows:

1. The primary test objective at the functional and system test level should be to show conformance to the system requirements.

   The testers should be able to "assume" that the system is reliable in its behaviour – ie that whatever it does, it does consistently and predictably.

2. The testing should be requirements based and should focus on correct processing and display of data.

3. The plan should aim for 100% requirements coverage.

Supporting Reviews

As a second priority, additional support can be provided to the test process by the use of reviews. The following reviews provide significant value:

1. The system level tests should undergo peer review. These reviews should include some developers.

   The developers participating in these reviews should apply their "white-box" knowledge of the system design and implementation to suggest better ways of running tests and additional test cases. In this way they can improve the system-level testing without significantly compromising its "independence".

2. The system design and code should undergo peer review. These reviews should include reviewers who are familiar with the requirements without being directly involved in the implementation. These independent reviewers could potentially come from the test group.

   The independent reviewers should concentrate on ensuring that the system requirements correctly flow through into the design and subsequent implementation.

Managing Software Handover

The project management must define completion of "coding" in terms of the ability of the independent testers to perform functional testing. This has two implications:

1. The testers must be able to "reject" code if it is inadequate for them to make meaningful functional assessments. Rejection of the code should not be seen as a failure by the testers.

2. Completion of code and unit test by the developers must not be seen purely in terms of software handover to test. If the software is inadequate for the testers to start functional testing, then they should be considered to have not yet completed the code and unit test task.

The Conflict Resolved?

Although the rules described here are simple, they provide a basic yet effective test strategy.

The strategy is effective because it focuses both developers and testers on doing what they do best, and targets the test types which give the most return for the resources expended.

Traditionally testers complained that they didn't get software which was robust enough for them to perform their planned and prepared tested. Using this strategy, they can reasonably expect to get robust software for test.

Developers traditionally complained that they were doing the testers' job. Using this strategy, they have a defined responsibility for delivering robust code and clear objectives for achieving this – a responsibility which few developers will dispute.

The division of tasks between developers and testers allocates test responsibility to the groups most skilled in the relevant area. Developers concentrate on the design and testers concentrate on the requirements.

Thus each group is spending time on the areas in which they are *most* suited.