

Test early, test often

John D. McGregor, Clemson University and Luminary Software LLC, U.S.A.

Abstract

Nothing does a software company's reputation more harm than poor quality. One significant contributor to delivered quality is how thoroughly the software is tested. In this issue of Strategic Software Engineering I will describe a test process that has a strategic impact on the organization.

1 INTRODUCTION

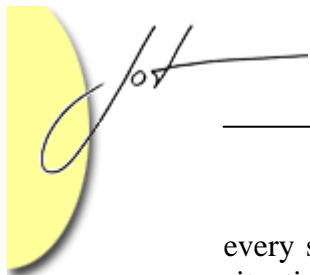
A colleague at Clemson University recently complained that his students could not write good test cases. At about that time, I worked with a very large development project that had just experienced the embarrassment of failing a customer acceptance test because multiple severe defects were discovered by the customer's tests. I thought it might be useful to explore the strategic significance of testing in software development.

When my client failed the acceptance test, that was not the worst of it. That client's client was integrating the failed software into an end-user product and missed their deadline due to not receiving the software on time. The failures were due to inadequate testing earlier in the life cycle. The costs of testing earlier would have been trivial compared to the penalties that were paid due to the failed tests at the end of the life cycle.

There are two arguments for not testing early and often. First, the "we can't afford it" gambit followed by the "we don't have enough time" excuse. Neither of these is correct and simply result from managers who fail to do complete analyses or any analysis at all. Even the most conservative estimates of the differences in costs of defect removal indicate that earlier is more cost effective.

In fact, Crosby claimed that quality is free [Crosby 79]. In the age of the Internet this relationship doesn't hold for every system. For software product lines it does. For web pages that present the current news for an hour it doesn't. The strategic approach is to examine the risks and rewards in every situation and make informed decisions based on data rather than on "its how we have always done it."

How much quality is enough? It is not enough to calculate defect densities. J. Adams found in 1984, that 1/3 of all faults only failed less than once every 5000 execution years. In an embedded control system in a car with say 1,000,000 copies around the world,



every such error will appear in one car every four days [Hatton 00]. We find this same situation in software product lines where an asset is used in multiple products all of which have large sales volume.

In the “dark ages,” testing referred to a development phase that followed the implementation phase. The Rational Unified Process now lists testing as a discipline that is active in every development phase [Kruchten 03]. In fact many development process descriptions now include activities from the testing discipline in every phase.

Figure 1 compares a profile of the test effort in an early testing environment as opposed to a more traditional late testing environment. The total amount of effort – the area under the curve – is less for the early testing approach than the late one. However, the real savings with early testing is not in the reduced cost of testing but in the reduced cost of repairing the software when a defect is found early in the development process. When found late the defect is likely to be fairly well tied into the product by dependencies and is much more time-consuming to remove than it would have been.

This begs the original question, what is a good test case? The purpose of testing is to find defects so a “good” test is one that finds a defect. To find a defect we have to look where the defects are and we have to be smart about how we look for them.

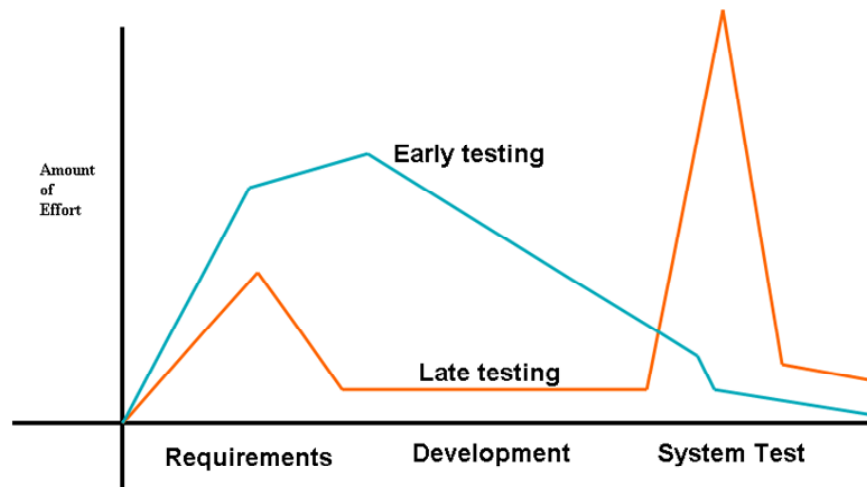
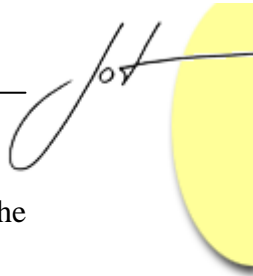


Figure 1 - Testing Effort Profile

First I want to consider the full range of testing and some basic concepts and then I will talk about some techniques for selecting test cases.

2 THE BIG PICTURE

Traditionally, testing is the execution of code using data that has been specifically selected. I want to expand that definition. Think of testing as “the exercising of an artifact to determine that it performs as it should.” That is a broad definition, but bear with me for



a page or so. Figure 2 shows a correspondence between development activities and the corresponding testing activities at each level in a software development process.

I use three techniques, each at the appropriate point in development. I will briefly describe each one and then focus on their similarities. These techniques are intended to provide comprehensive coverage of the complete software development process.

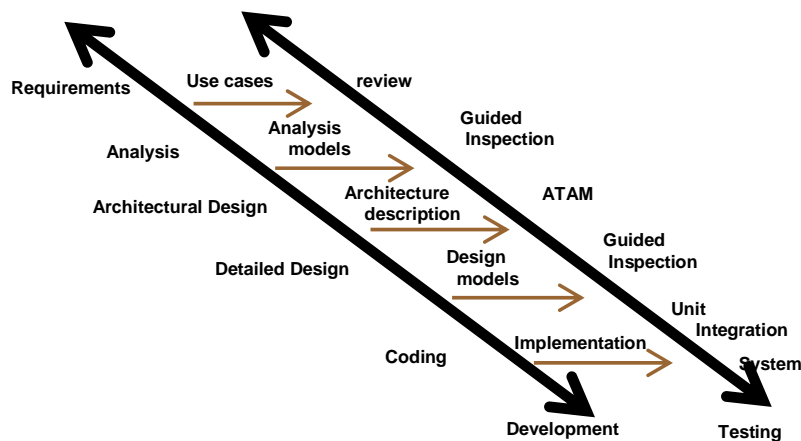


Figure 2 - Test activities in the development process

Guided Inspection

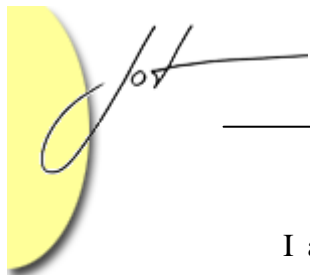
Early in the life of a software product there is no code to execute but there are models – requirements models, analysis models, architecture models, and others. Guided Inspection is an inspection technique in which a reviewer examines a model under the guidance of scenarios specifically chosen to determine if certain defects exist. The reviewers determine how the model handles the scenario by tracing through the content of the model and use expert judgement to determine whether the trace is “correct” or not.

The SEI has a method, ATAM[™], devoted to the equivalent of a guided inspection of the architecture [Clements 02]. The inspectors work with all stakeholders to determine the areas of the architecture that have the highest priorities among the stakeholders. Scenarios investigating those areas are used to estimate how well the architecture meets its requirements in those areas. The inspectors use their knowledge of architecture to inject additional scenarios concerning typical defects in architectures in general. This is a very effective technique for identifying defects during architecture definition.

Simulation

Some modeling notations are so complete that execution of the final product can actually be simulated using the model. Most often done using an architectural model that has been defined in an architectural description language or a detailed design model in a formal design notation. A simulation steps through actions in the definition to exercise a variety of features of the system being modeled and to examine its behavior.

[™] ATAM is a trademark of the Software Engineering Institute.



I am currently using the Architecture Analysis and Design Language (AADL) to represent architectures [AADL 07]. I also use the Architecture Description Simulation (ADeS) tool to simulate portions of the behavior (note this tool still only simulates a portion of an AADL architecture) [Axlog 07]. This allows me to select scenarios and get some simulated executions of those scenarios. The people producing ADeS started by simulating the behavior of processes and threads. This is a big help to the inspectors since this is detailed and intricate behavior.

Code Execution

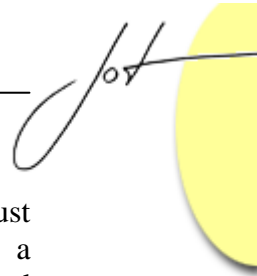
This is the activity most often thought of as testing. Data values are selected and used as input to some size of code unit. The data may be entered via the system's GUI or it may be injected by a test GUI or a specially designed harness. The execution of the code produces a trace through the software under test as well as some type of final result. This activity requires an "oracle," a source of the truth, such as a domain expert or some other means of determining whether a test has found a defect or not. In addition to determining whether the final result is "correct" the trace may be examined to be certain the expected path was followed.

Early testing has been made less painful by several tools. The XUnit family of unit test harnesses supports unit testing in a variety of languages and environments [JUnit 07]. The TPTP project in the Eclipse Open Source Software organization integrates JUnit into the Eclipse environment and provides other tools such as GUI testers. There is still time for "early" testing even after some end user functionality is developed if the development team is using an incremental development approach. "Early" refers to the lifetime of the software under test, not a particular development activity.

Testing Perspective

The actions described above all require personnel that can use them effectively. Testing is an activity with a certain kind of attitude or at least the people who perform tests should have a certain attitude. I call it the testing perspective. A successful tester is:

- **Systematic** – Testing is a search for defects and an effective search must be systematic about where it looks. The tester must follow a well-defined process when they are selecting test cases so that it is clear what has been tested and what has not.
- **Objective** – The tester should not make assumptions about the work to be tested. Following specific algorithms for test case selection removes any of the tester's personal feelings about what is likely to be correct or incorrect.
- **Thorough** – The tests should reach some level of coverage of the work being examined that is "complete" by some definition. Essentially, for some classes of defects, tests should look everywhere those defects could be located.
- **Skeptical** – The tester should not accept any claim of correctness until it has been verified by testing.



My reason for identifying these characteristics is because many people, not just “professional” testers, participate in testing activities during the development of a software product. All the stakeholders will help define scenarios during an ATAM and many domain experts will participate in guided inspections. All of these participants need to adopt the testing perspective during these activities.

3 BASIC CONCEPTS

In this section I will run through a few basic concepts.

Fault Model

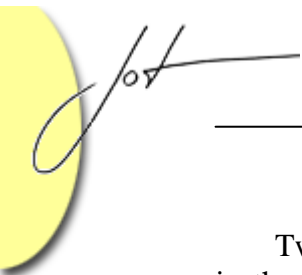
Software execution occurs in a context. Testers are examining what can go wrong with the software in that context. For example, certain faults can occur in a real-time system that can not occur in a batch oriented data processing context. Essentially, a fault model is a representation of what can go wrong in some context. When faced with a testing assignment, the tester should first find a fault model that covers the context in which the assignment is located (or create such a model) and then begin to select test cases. They can create test cases that will examine the work in a way that has the chance of failing if a defect like the one in the fault model exists in the work under examination. The work at IBM on Orthogonal Defect Classification provides a starting set of fault models [ODC 07].

Test case

A test case is the description of one exercise to be applied to the artifact under test (AUT). The test case can be represented as a triple <pre-conditions, input, expected output>. The pre-conditions are constraints on how the test is conducted. For example, suppose we are testing a toggle switch function, one that moves from one state to another whenever it is activated. It is necessary to specify the state the switch is in at the start of the test in order to know what state it should be in at the end of the test. I will talk more about the input data for a test case in just a few lines. The expected output specifies what the exercise of the AUT should produce. Any deviation from this expectation is a failure.

Test coverage

When a set of test cases has been applied to an AUT, certain parts of the AUT have been exercised and certain parts have not been exercised. The part that has been exercised is said to have been covered. Part of being systematic is to select test cases so that certain patterns of coverage are achieved. For example, test cases could be chosen so that each exit from a decision point is exercised. This is referred to as “branch level coverage.” When this type of coverage has been achieved, the tester knows that certain types of defects don’t exist. A number of development environments and test tools come with tools that will assist in measuring coverage.



Two testers can develop the same number of tests but there can be a large difference in the effectiveness of the two sets of tests. By judging each set as to the coverage it achieves, we have a standard against which we can evaluate the two sets of tests and we can instruct the tester with the least effective tests how to improve.

Coverage also lets the tester develop an efficient set of tests. Two test cases that cover exactly the same trace through a program will only uncover the same set of defects. The tester can eliminate one of the two test cases without affecting the effectiveness of the set of tests but it will improve the efficiency of the test set.

Test suites

A test suite is a set of tests that are grouped together to achieve a certain level of coverage. This is a convenience device that supports the reuse of test cases. A particular test case can be used in multiple suites and applied whenever a specific level of coverage is desired.

- Functional Test Suite - This test suite is made up of test cases that achieve some level of coverage of all the specified functionality for the AUT.
- Structural Test Suite – This test suite is made up of test cases that achieve some level of coverage of all the code statements or execution paths in the AUT.

Once the AUT has passed the test cases in the functional test suite, the tester knows that the AUT can successfully perform its function. Once the AUT has passed the test cases in the structural test suite, the tester knows that there is little likelihood that the AUT does anything that it is not supposed to.

4 STRATEGIC USE OF TESTING

Testing is often seen as a sink for resources but it can have a strategic impact on the organization. According to data presented by a number of companies, early testing can provide a cost improvement of as much as 100 times over the practice of relying on system testing that occurs after integration.

Value Added Testing

Boehm points out that many software engineering techniques are value neutral [Boehm 05]. That is, our techniques process all information the same regardless of its potential benefit to the organization. The techniques I have been describing in this paper are value aware techniques.

All of these techniques are scenario-based. In each case the scenarios are selected with input from stakeholders. The criteria for selection are risk and criticality. Using this type of approach, overall, the test suites will add much more value to the product than tests selected by algorithm. Once the basic test case is selected, specific data for the test can be randomly generated.



Test Driven Design

Test-driven design (TDD) is an extreme view of the “test early, test often” approach. Tests are written, code is written, and then the tests are run to determine that the code faithfully represents the requirements as captured in the tests. This is repeated for slivers of functionality. This provides a very early set of tests of code. My concern as always is that TDD does not take a sufficiently comprehensive view before diving into details. TDD does not produce an architecture.

Test-driven design is only possible if the tests are automated. It is more possible if the testing is embedded in the IDE. JUnit is integrated into the Eclipse Java Development Tools in a way that makes running unit tests as easy as running the program.

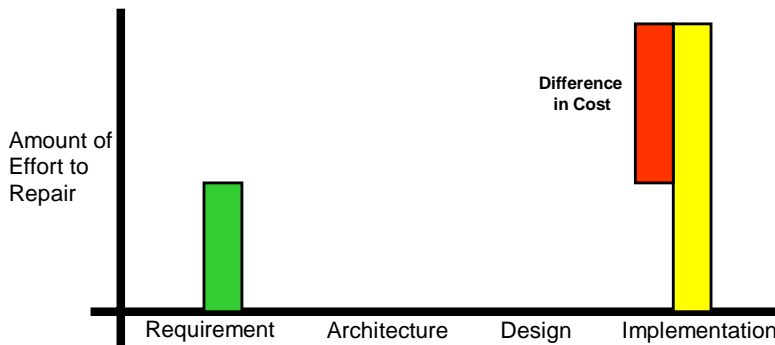


Figure 3 - Cost Comparison

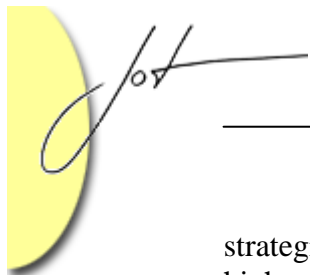
Process Improvement

The test activities provide information that can be very useful in process improvement efforts. Test reports identify failures. These reports lead to developers identifying causes of these failures. Process engineers can then classify the “live range” of each failure-causing defect. By live range I mean the period between where a defect was introduced and the point at which the defect was identified – the range of development activities in which the defect was alive.

Given the live range for a set of defects the process engineer attempts to determine why the testing activities inbetween the ends of a defect’s live range failed to identify the defect. New activities may be added to the process and existing activities may be changed. The activities may not be testing activities. For example, the requirements activity may be changed to eliminate defect causing behavior. New, more rigorous levels of coverage may be imposed.

5 SUMMARY

The test discipline can be a constructive part of the software development process. It provides critical feedback about activities conducted earlier in the development process as to problems in the specific software being tested and potentially in the development process itself. Changes to the testing activities in the development process can have a



strategic impact on an organization in the form of increased productivity, lower costs, and higher quality.

REFERENCES

- [AADL 07] <http://www.aadl.info>
- [Axlog 07] http://www.axlog.fr/aadl/ades_en.html
- [Boehm 05] Barry Boehm. *The Economics of Software Quality*, Motorola Quality Workshop, 2005.
- [Clements 02] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley 2002.
- [Crosby 79] Philip Crosby. *Quality is Free*. New York: McGraw-Hill
- [Hatton 00] Les Hatton. *The Economics of Software Testing*. EuroStar 2000.
- [JUnit 07] www.junit.org.
- [Kruchten 03] Philippe Kruchten. *The Rational Unified Process: An Introduction*, Addison Wesley, 2003.
- [ODC 07] <http://www.research.ibm.com/softeng/ODC/ODC.HTM>

About the author

Dr. John D. McGregor is an associate professor of computer science at Clemson University and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-based software engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). Contact him at johnmc@lumsoft.com.