SAND96-1228C

CONF-9606129--1

Software Testing Process Improvements[†]

Dr. Dwayne L. Knirk

Sandia National Laboratories
P.O. Box 5800, MS 0638
Albuquerque, NM 87185-0638

Contents

1. Introduction

Software process improvement has become a popular pastime, for a variety of reasons. The Software Engineering Institute's summary of experiential data, which resulted in the Capability Maturity Model, has now had considerable corroboration.

There are nearly as many software processes as there are combinations of developers, users, and products. Similarly, there are probably as many software process improvement approaches. However, the meta-process for performing process improvement is quite straightforward.[1] Processes can be represented by a small number of abstractions, with variety supplied through implementation details. The scheme for improvement is almost self-evident: figure out where you are now, use a software process maturity guide to identify shortcomings, plot a change in a direction to eliminate a shortcoming, and go for it.

This paper won't dwell on the meta process and its enactment; we simply assume one is in place. Rather, we consider some ways to improve the testing aspects of your software process. These may be changes in *what* you do for testing as well as in *how* you do it.

You wouldn't want to make all these changes at once. On the other hand, you should want to make some of the changes. Which changes would be most suitable for you? Which ones should you make first? This paper describes some of the factors you should consider in developing a good strategy for improving the testing processes in your organization.

2. Software Testing Context

Testing is a process of executing software under specific controlled conditions, observing or recording the results, and making an evaluation of the software. This standard definition, paraphrased from the IEEE Std. 610.12, IEEE Standard Glossary of Software Engineering Terminology, describes only the most commonly visible step in the testing process. Just as the software to be tested does not simply materialize, neither does the testware[2] you need.

2.1. Purpose of software testing

The purpose of software testing is to reduce our ignorance about the operational properties of our software. No matter how well or how poorly we developed the software, defects of omission and of commission will most likely be present. Somewhere, sometime, when you least expect it, those defects will cause failure. Testing is the last chance to find these defects so they can be fixed before the customer experiences operational failures.

Testing allows an evaluation of the risk of shipping software in its present form. The process for getting the information needed for this evaluation is to exercise the software:

(a) to demonstrate it does what is expected, and nothing else, and

(b) to expose whatever bugs may be present.

We may regard positive test results as evidence for (a) and negative test results as evidence for (b). We must gather as much convincing evidence as we can for this evaluation within time and budget constraints.

For most commercial and industrial software, exhaustive testing is impossible. In its stead, we create test cases by selectively sampling the variety of situations in which the software may be used. Because the number of sample situations is so small compared to the number of possible situations, the few test cases we create must be good. We can't afford to waste our testing resources. Every test case must show something

DISTRIBUTION ~~OF THIS DOCUMENT~~ IS UNLIMITED

[†] This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

MASTER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

about the software that no other test case shows.

The testing we do answers the questions "Does it work?" and "What doesn't work?" The testing we don't do reduces our confidence in the software's correctness. Thus, it is as necessary to know what has not been tested as it is to know what has been tested. Without a measure of overall "goodness" of testing, we cannot use test results to quantify the goodness of the software. That is, even if the software passes a poor set of tests, you remain ignorant of its operational qualities and unable to evaluate its adequacy.

2.2. *Prerequisites for testing*

Testing is not the only practice for detecting and removing defects in the software, but it is the last one in line before delivering software to your customers. The value you derive from testing depends on several other factors in the software development environment. The adequacy of those factors does not guarantee success in testing, but their absence or inadequacy certainly diminishes your ability to accomplish testing. In particular, software testing success depends directly on the following things.

Documented software requirements. Software requirements that are not documented must be remembered from the beginning of the project to the end of testing. If they are not remembered, how can they be checked? It is unreasonable to expect any development team to keep all the requirements in their head. Documented requirements provide the minimum checklist for completeness of testing.

Design for testability. Besides the external requirements for various software characteristics, requirements for testability should also be met by software design and development. If no thought for testing is taken during development, the testing job can become much more difficult.

Code inspections. Numerous comparative studies show that code inspections can detect up to five times as many defects as testing. Those studies also show considerable savings of the time and effort needed for detection by inspection compared with testing. If code inspections are not performed, then testing has to contend with the presence of many more defects. It is a common situation that occurrence of one failure during test execution often hides other possible failures. Failures in testing one situation may prevent testing other situations until the defects are fixed. This impedes the rate of test execution and increases the number of times the software must be reworked.

Code version control. In the normal repetition of fix, build, and test cycles, multiple versions of code are being generated and tested. Testing will be completely ineffective if the wrong version of code is being tested. Also, you need to be able to back up from code fixes which prove to be wrong. The original version must be kept for starting over. Finally, if multiple versions of the code are currently in the field and you must maintain them (under warranty or service contract), you must keep copies of source code for at least those versions. You must also be able to propagate fixes into each version having the same bug.

Test entry criteria. A simple condition for all software to be tested is that there be no compiler and linker error messages. Warning messages may be acceptable if there is a known reason that they should occur in building the test executable code. Lint-like tools and procedures like code inspections should be applied before you start spending your testing resources.

Test execution environment. Dynamic testing requires some kind of execution platform for the software. The primary requirement for a testing platform is that the software under test can be manipulated and observed in a controlled environment. Among the platform alternatives are these:

In-house target system. This alternative provides an exact reproduction of the software's operational environment. This is easy for a software-only application operating on a commercial computer system, but it can be more costly for an embedded application.

Simulator/stimulator. This alternative uses the target computational hardware, but simulates the rest of the system to produce equivalent interactions with the software. Stimulations may be applied from completely outside the target system. Capture/playback tools, on the other hand, intercept input/output paths in the internal software infrastructure.

Emulator. Emulators are appropriate for deeply embedded software. They are usually interpretive and mimic the actual target code processing on a different computer such as the development computer. They offer the same visibility into code operation as would an on-line debugger on the real target system.

3. A Process Framework

For the discussions in this section, please refer to

Figure 1. It illustrates a framework for software process in terms of various technical work products Note that the diagram does not focus on activities and their sequencing. Rather, it focuses on work products containing various kinds of software information and the content dependencies among these.

The activities in the development and testing processes are *the lines connecting the work products.* It seems easy to understand a static diagram of interdependent work products and to envision them coming into being and maturing as time passes. The activities around these work products, on the other hand, are multiple, they are dynamic, and they can have multiple occurrences as well. For example, a Software Design is created to be consistent with a current specification. Some prototyping and performance analysis may be performed during its creation. It is reviewed for testability, implementability, and completeness against requirements, possibly leading to its update. And as requirement change, many of these activities may be repeated at a later time. All these activities serve but one purpose — to maintain consistency between the Software Design and the work products.

3.1. Work products

Labels across the top of Figure 1 name various work product groups. In a waterfall process, these would correspond to actual phase endpoints.

There are two distinct dependency threads in Figure 1, one for software development and one for testware development. Both start in the Isolation phase and pass independently through the Formulation and Implementation phases. They rejoin in the Evaluation phase on test execution results (trace and pass/fail information). Test execution requires both the software and the testware to be available. Conversely, however, test planning and the design and preparation of testware can be performed simultaneously with, and somewhat independently of, the design and implementation of the software.

The activities that establish these dependency threads from left to right in the figure are just the following typical development steps:

- *analyze problem requirements*
- *design and specify the software's behavior*
- *design and specify code and data structures*
- *encode data and unit algorithms*
- *test and evaluate system*

The vertical and diagonal lines correspond with reviews (inspections) and working discussions between developers and testers.

Here is a summary of the work product phases.[3]

Conception. The functions to be accomplished by coordinated actions of the software, hardware, and bioware must be understood. That understanding is often captured in a concept of operations document. That document should clarify the scope of the application problem and the assumptions granted to an intended solution. It places the software component in perspective of its intended use.

Isolation. Focusing on the software in isolation is a specification document identifying everything the software should do. Ideally, this specification is *predictive*. That is, it can actually be used to answer

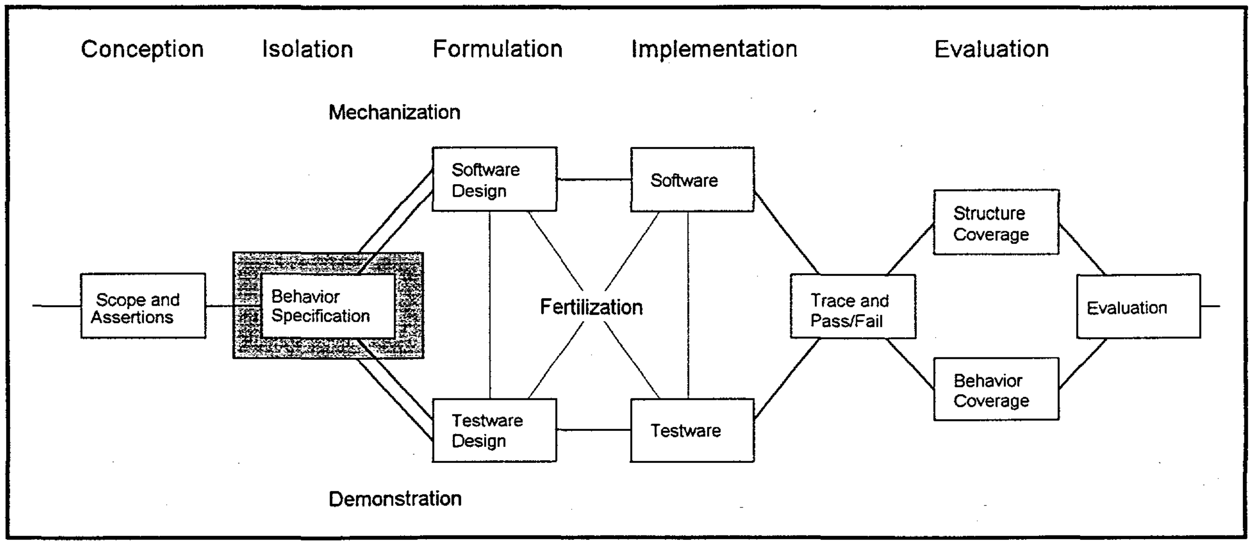

Figure 1. Work Products of the Development and Testing Processes in the Software Project

questions of the form, "What does the software do when P happens in situation Q?" Such a specification is called a Behavior Specification. It is sufficient for designing the software as well as designing the testware.

Formulation. Dependencies on the Behavior Specification split into upper and lower threads. The upper half of the diagram illustrates our familiar approach to the *mechanization* problem. Software is designed and developed as a mechanism for providing some computing capability to a user's job. The lower half of the diagram illustrates a parallel approach to the *demonstration* problem. Testware is designed and developed, not as a deliverable computing capability, but as an exhibitor of the software's intended capabilities. This emphasizes the analogy between software design as a problem in mechanism creation and testware design as a problem in demonstration creation.

Implementation. The principal information dependencies continue separately through the upper mechanization thread and the lower demonstration thread. Software is produced in accord with the software design. Software is embodied in code and data files. Testware is prepared in accord with the testware design. Testware should also be embodied in code and data files, though it is often deposited into documents.

A significant synergy occurs as cross-fertilization between the two threads. This is indicated by the vertical lines in the diagram. Concurrent design of the software and the testware will enhance testability of the software as well as software manipulation by the testware.

Evaluation. Evaluation is split into three distinct steps. The first step is to bring the testware and software together for test execution. The software and testware are set up, test cases are executed, and observations collected. Traces of control and data flow are produced during the tests. Test case results are checked against the expected results to determine whether the software passes or fails. Testware "fails" if it doesn't do what a test is supposed to do, just like software "fails" if it doesn't do what the code is supposed to do.

Structure coverage of the tests is determined from various characteristics of these traces: all statement execution, all branch execution, all condition values, all paths through decisions, etc. Test case pass/fail results are tallied against the test objectives to which the test cases correlate.

Based on the goodness of test objective coverage and the extent of structure coverage, an overall evaluation of the software is determined.

3.2. Testing Process Elements

A more comprehensive definition for the testing process than the one in the IEEE glossary is this: the development and application of testware by which software can be exercised and its behavior evaluated against expectations. This process includes

- *determining requirements and designs for the testware code, data, and procedures;*
- *creating, acquiring, or simulating representative situations the software must handle;*
- *executing and observing the software's behavior; and*
- *evaluating testware and software goodness.*

These are the activities occurring across the bottom thread in Figure 1.

IEEE Std. 829, IEEE Standard for Software Test Documentation directs creation of several significant work products for testing. To accomplish effective testing you will find yourself doing the work they describe. To accomplish effective retesting of modified software, you will save time and effort by reusing the information they capture. We paraphrase the standard to identify the following work products: test objectives, testware designs, testware, test logs, and test coverage. They are described below.

Test objectives. These are statements of all observable behaviors which the software must exhibit during operation. Software requirements documents are notoriously incomplete in this respect. As a tester, it is up to you to

1. *identify the required inputs to be provided;*
2. *indicate the particular situations in which those actions can be executed, including any dependencies on previous actions;*
3. *specify how to initiate the behavior; and*
4. *describe the outputs and other changes which may result from the behavior.*

Test objectives catalog every operational thing to be shown about the software. The core of this catalog constitutes a Behavior Specification of the software. Testing to these objectives will build confidence that the software does what it is supposed to and does not do what it is not supposed to.

Testware design. No component as complex as software is simply built. Rather a design for the mechanism is created first. The mechanism's design identifies software elements that already exist as well as new software elements to be created. A scheme for interlinking the elements into the finished software is settled, and a method for controlling the capabilities of each element and integrating their behavior is implemented. A software design is complete and correct if all required behaviors of the software can be produced by the mechanism.

Analogous statements can be made for testware. First, a design for the demonstration is created. The demonstration's design identifies required test data, support code, and tools that may already exist or that may have to be created. A scheme for interlinking the tools and support code with the software under test is settled, and methods for stimulating the software with the test cases and for observing the results of the tests are determined. A testware design is complete and correct if all required behaviors of the software can be demonstrated by the testware.

Testware. Testware includes test cases, test procedures, and a test execution environment. Test cases consist of data values and event sequences that are needed for demonstrating individual behaviors of the software under test. It is important that there are expected results for executing a test case. Inherent in testing is the satisfaction of an expectation, since it is a failure when something other than the expected occurs.

Correspondent with the test cases are procedures for their execution. Test procedures identify required sequences of the individual test cases, specified timings of event sequences, durations, and overlaps, and the manipulations required to set up, initiate, and observe the tests. These procedures are frequently performed by automated test case execution tools.

Test logs. A test log is a time-stamped record of test execution. It is a by-product of producing test execution traces and test case pass/fail. When an unexpected result is encountered, it is referenced in the log. Later, the log can be examined to see what was done to the software leading up to the failure. Test logs are valuable debugging aids.

Test coverage. Test coverage is a measure of "thoroughness" of testing. It contains two parts: coverage with respect to demonstration of the test objectives and coverage with respect to exercise of the software mechanism. Coverage with respect to demonstration of the test objectives requires tracing between test cases which "pass" and the test objectives which are met by the test case. Coverage with respect to the software mechanism requires tracing the software's internal control logic flow and data flows and determining which flows have been exercised by the test cases.

4. Process Improvements

Do you need to make changes in your testing process? The answer may be yes if your software testing is planned, managed, and practiced like this:

- *Testing means running the software continuously between the time software development concludes and the time the product ships.*
- *A test case is whatever the tester has decided to do next.*
- *A test case passes if the tester is happy with the results.*
- *The software fails frequently in operation.*

If we agree what we should be doing, and we admit what we aren't doing, the question is "How do we get from where we are to where we would like to be?" The rest of this section describes six potential improvements to the testing process:

- *Interweave test and development processes*
- *Automate test case execution*
- *Establish testing objectives*
- *Measure actual test coverage*
- *Build test coverage systematically*
- *Document failures and fixes*

Attach no significance to the order of presentation. Different situations would recommend different sequencing of improvements. This is discussed in Section 5.

A general process improvement philosophy is that of capturing work results in some kind of document, even ASCII text files. The cost justification for this is very strong in a maintenance-oriented organization. It may be cheaper initially just to create test cases on the fly. But every time a bug is found and fixed, or the software is enhanced or adapted, you must perform that test creation job again from the beginning! You faced either paying the cost of repeating work you have done before, or avoiding the cost by simply not re-testing. (Don't bother, it was a small change...)

Neither of these alternatives compares sensibly with documenting the test cases when they are created, augmenting them when changes are made, and then reusing them for every subsequent release.

In addition, the "create and apply on the fly" approach forces developers and testers to work in isolation. Whatever one learns is not transferable to another unless it is captured in a reusable form. Furthermore, whatever one does cannot benefit from the knowledge of others through inspections unless it is captured in a readable form.

In the following subsections, each process improvement is described in terms of its purpose and your likely benefits and costs. Each will cause the establishment of concrete, reusable testing work products. Each can and should be considered for implementation at the system testing as well as the unit testing levels.

4.1. Interweave test and development processes

If you defer all testing work until the code is complete, there is rarely enough time to execute and evaluate those test cases that occur to the testers. There is never enough time for the intentional design and preparation of adequate testware and for the evaluation of test goodness. Thus, except for a list of test failures, there is no support for any quantitative evaluation of software goodness or acceptability.

In addition, by deferring all test-oriented issues until the code is complete, there is little possibility of actually designing and implementing testable code. The software staff provides themselves with no more facility for manipulating and observing their software's behavior than they provide their external customers.

In this process improvement, you partition the current testing work into two separately accountable activities: (1) testware design and preparation, and (2) test execution and evaluation.

Schedule the first activity concurrently with software design and implementation. Test cases and test procedure(s) should be the work products from this activity. These work products may be in either electronic or paper form, but they must be explicit, concrete, and shareable between staff members. The intention here is to try to use the specification information as soon as possible for testing. First, this is an operational "review for testability" and can have a salutary effect in minimizing the propagation of misconceptions into the software design work. Second, doing this work now provides concrete interpretations of the specification for reference by the mechanism builders. Third, it furnishes enough detail for really planning the necessary testing effort.

The second activity is essentially the usual execution and evaluation work. However, the testing work performed earlier does not have to be done now. How you perform tests is unaffected. Use whatever approach you used in the past, whether manual or automated, but do it now with documented cases and procedures. In addition, scripted procedures can be annotated to record progress through the testing and incidents that may be software failures.

If you don't currently prepare test case and procedure documentation, you will feel this is an increment in the cost of testing. Actually, you can expect a systematic approach to improve your test cases and reduce the costs you incur for defect delivery. Even if, in the worst case, you find no more defects than you would have, the considerable value of preparing test cases and procedures is apparent on the very next release of the software. All the mental testing work done before to devise the cases and procedures is now available for reuse. Some changes may be needed to account for enhancements or changes to the software, but you are not starting over on testing. Only the test execution and evaluation activities need to be repeated.

Imagine if we just threw away the "useless" source code once the executables were compiled and shipped. Talk about starting over! Yet that is just the effect of "create and apply on the fly" testing.

4.2. Automate test case execution

Another improvement is to enlist the computer itself to assist in the test execution and evaluation activity. The automation of test case execution does not directly help you test any better, but it does help you test faster, more consistently, and with minimal human involvement. The time you save in repeated test execution can be invested in other improvements that do help you test better.

There are two distinguishable cases based on the predominant interfaces to the subject software system or component.

API interfaces. For software components bounded by command lines or application programming interfaces (API), the primary behavior to be tested is *functional.* That is, use of the software elicits one or more stimulus-response interactions between the software and its environment (operating system, other hardware, communication systems, or client/server interfaces).

GUI interfaces. For software components bounded by graphical user interfaces (GUI), the primary behavior to be tested is *operator-software dialog*. The software goes through various sequences of interactions at the GUI. The net effect of a sequence is to prepare one functional input to the rest of the system or to communicate one functional output from the rest of the system.

Many software systems, even if not designed this way intentionally, can be tested successfully by combining these two interface perspectives.

Automated dialog execution. Many GUI development platforms can be supplemented with capture/playback tools. These tools work only with inputs and outputs across the GUI itself. They can record all inputs and outputs during a test, then later they can repeat the test identically by re-entering all inputs and comparing all new outputs with the original. Discrepancies suggest possible failures of the software.

Capture/playback tools do not help with testing at the other interfaces because they do not capture and playback data and signals at those kinds of interface ports.[4] Nevertheless, capture/playback tools can provide considerable benefit. The first time they are used, all GUI test procedures are followed and the capture function collects all GUI inputs and outputs, including timing information. This is the just like the manual testing job. However, all successive times these tests must be run, the playback from the initial testing is completely automatic.

Automated function execution. Any development platform can be supplemented with standardized tools to harness code elements at their programming interfaces. Figure 2 illustrates one kind of harness you might use. The idea is to create a controlled environment around the test subject.

A driver is code that sets up unit input data in its executable form and actually invokes entry points to execute the test. It replaces the potential callers of the unit under test. A unit test driver can be largely standardized so individual developer has very small amount of extra work in adapting it to any particular unit.

A stub is simple code that has the same invocation interface as a real subordinate for receiving and returning calls or messages from the unit. It provides substitutes for services the unit requires in accomplishing its specified behaviors. It may be so dumb as to record only that "Kilroy was here," or it may record or print the calling argument values. A smarter stub might let you specify what kind of response to make.

Previously tested code as well as operating system services may be used in establishing the control and observation environment for the unit under test.

4.3. Establish testing objectives

Another improvement is to expand your behavior specification activity to include the production of a catalog of test objectives. These objectives can be included in tables in an existing software requirements specification document, or they can be captured in a

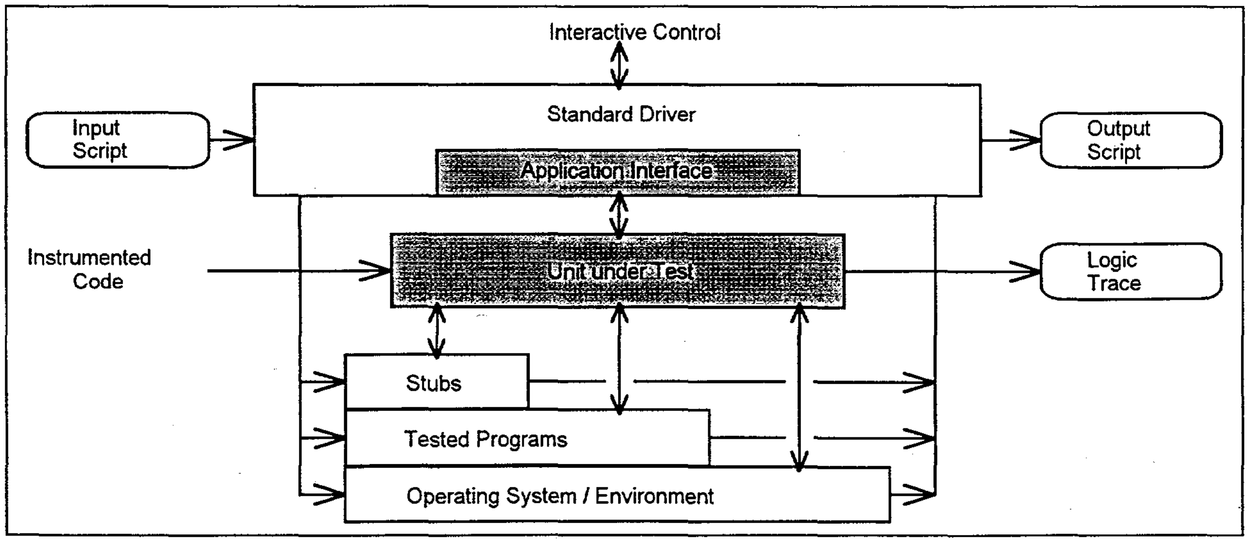

Figure 2. Standardized Test Execution Harness

separate documentation form.

It is a common practice in commerce and industry to jump directly from a statement of the application problem to an architecture of a solution. Time is not available for determining that behavior offered by the solution meets the problem's requirements. [Recall the comments in subsection 3.1 about the behavior specification.]

Test objectives are often created very early in the software design work, but then they are not documented anywhere. A detailed requirement statement may say "User errors when entering recipe names will be caught and an error message displayed." The designer begins clarifying this by asking questions like

- *Does the message come up after the list is finished or immediately after an erroneous entry?*
- *Does the message require user acknowledgment?*
- *What should the message say?*

Answers to these questions describe specific observable behaviors of the software. None of this information is internal software design.[5]

Given these specifics, internal software designs may be created and appropriate tests for them conceived. Each specific statement is a test objective. These statements should be collected into a catalog of test objectives. This catalog identifies every observable function, the valid and invalid subdomains of possible inputs that must be responded to, and all operation modes and mode changes. By creating a test case for each test objective, you know you are demonstrating every operational requirement of the software.

It is too common that such specific behaviors are not written down for the software requirements. They are conceived and used by the software designers. Later, they are recalled and explained to manual writers and customer trainers, or they are left to be discovered by experimentation. Testers cannot discover them until after coding is complete. If they were captured in the beginning, much repetition and miscommunication could be avoided.

In addition, by having test cases for each test objective, and test objectives for each operational requirement, you would be far more likely to expose possible software failures before your customer finds them. You would also have a much better understanding of the full scope of the testing that needs to be performed.

4.4. Measure actual test coverage

This improvement implements a specific activity for evaluating test results. It is separate from the test execution activity that produces those results. Evaluation has two components: one is an evaluation of the software adequacy, the other is an evaluation of the testware adequacy, each measured against its own criteria. Simplistically, we must check that the software not only passes all tests, but that those test cases represent all application situations in which the software is to be used. (We don't require the software to do anything meaningful outside its intended scope of application.)

Software adequacy. Software is adequate if it produces correct responses for the situations in which it will be applied using all parts of its mechanism. To determine if correct responses are given, you examine the actual outputs for acceptability, often within some tolerance range. While this may sound obvious, it has been overlooked by testers intent on achieving usage of all parts of the mechanism.

To determine if all parts of the mechanism are used, you examine execution traces of code control and data flow with respect to various criteria like statement coverage, branch coverage, and def-use path coverage. (A def-use pair is a statement assigning a value to a variable followed by a statement later in the path that uses the value from that assignment.) When all behaviors of the software have been exercised, no parts of the mechanism should be unused. If they are, your understanding of the required behaviors may be incomplete.

Testware adequacy. Testware is adequate if it correctly sets up enough samples of all the situations in the software's intended application, and then forces the software to demonstrate the required responses for each of them. To determine this, you must trace, or correlate, each test case to the test objective(s) that its execution demonstrates (for example: function F3, in state S7, at low boundary of the input variable V12). The second step is iterated over all test objectives: You collate the pass/fail decisions for each test case that demonstrates a given test objective; and if they all passed, you declare the test objective met. Testware adequacy is an evaluation of how "thoroughly" the black-box requirements are satisfied by the software.

When both of software and testware adequacy are high, your confidence in the goodness of the product can be high. When either of these is low, your risk of delivering unfound defects in the code is much higher. There may be no defects, but without establishing consistency between the behavior specification, the execution traces of the code implementation, and the test objective demonstrations, you simply cannot know.

Low coverage is likely to be followed by substantial bug reports from customers. Over time, you can correlate measures of test coverage with how much field support is needed for a software release. Then the measurement of coverage supports intelligent business decisions about when a software release is ready to ship and what costs to expected for its support.

4.5. *Build test coverage systematically*

This improvement also has two components. The first is refining the granularity of test objectives and distributing them across *levels of testing*. The second is *systematizing the design of test cases* for a set of test objectives. These operate together to increase the confidence in the software for a given investment of effort. Conversely, they reduce the investment of effort required to gain a given level of confidence.

Levels of testing. The futility of doing all testing on the finished system is an old song. Many would claim that they don't do this, that code units are tested by their developers. But, if developers test their own code and have no evidence afterward of what was done or how adequate it was, can a system tester afford to assume anything about the units in the system? No. They must perform unit testing in the system context, which is costly in time and quality. They may be repeating work that was already performed. The cost to locate and fix a unit's bug during system testing can be up to one hundred times as much as the cost during unit testing because unit testing in the system context is very inefficient. And every dollar spent on unit testing at system test time reduces the resources that can be spent on actual integration and system testing issues.

The point of this improvement, then, is to test software at three levels. You must allocate testing objectives to the lowest level at which they can be demonstrated, and then test for them at that level. You must build your confidence in the correctness of the system with the evidence you gather as you build the system.

Unit. Each software unit of work should have its own testware elements: test objectives, test cases and procedures, execution framework, etc. The unit's developer should test whatever behaviors of the unit can be tested. Simple tables of test objectives and test designs in files are sufficient documentation. Automated test execution scripts and code segments can provide the rest.

Integration build. Units are combined into higher level integration builds. Depending on the product's architecture, builds could be object clusters (or some equivalent), functional layers, transaction centers, or functional areas. They could be functional subsets of the entire system. Test builds for those things that could not be tested in the individual units, such as internal interface compatibility, internal state sequencing, and non-interference between the build components. Often, some end-to-end functions of the software can be tested as well. Structurally, one should try to execute all subordinate unit calls and call slices for each unit.

System. When the complete software system is available, you need to test for compatibility of physical interfaces, temporal sequences of execution, safety, robustness, and end-to-end functions of the software. But test only for those things that could not be tested in the various builds.

If you accumulate your evidence of software correctness as you test across levels, you reduce the total amount of work required to achieve a given level of confidence in the correctness of a complex software product. Planning a leveled strategy for a software product may also provide valuable insight into more testable software design alternatives. It certainly simplifies retesting during maintenance.

Systematizing test design. The objective here is to incorporate well-known specification-based test design techniques into your testware design activity. Expand on basic functional testing to include equivalence class testing, boundary value testing, partition testing, and state transition network traversals. Contrary to popular belief, developers and testers are *not* born with these skills. They need to learn them, not to discover them on the job.

Improved test design provides a much stronger set of test cases for exhibiting possible failures. Empirically, we find the increased coverage of the test cases can dramatically reduce the number of defects remaining in the software when it is delivered.[6]

Here is an effective process combining specification-based test design with structural coverage to the benefit of both.

1. *Apply black box test design methods to create test cases that will demonstrate required behaviors.*
2. *Execute those test cases with code that has been instrumented for code control and data flow tracing.*
3. *Remedy discrepancies. In particular, if code is missing, some tests of behavior will fail. If code has behaviors besides those required, then the traces will not completely cover the code structure. Either code should be removed, or documented behavior should be expanded. Upon code or specification change, iterate to self-consistency.*

4.6. Document failures and fixes

This improvement applies the knowledge of failures in the past to reduce the number of defects in the future. It can be leveraged through two mechanisms, reducing the number of defects inserted into the software during design and implementation, and sharpening the ability to design useful test cases to catch those defects. This improvement would supplement a similar feedback cycle that may already be present in an inspection process.

This improvement effectively recycles ordinary waste from the debugging activity. When failures are demonstrated during test execution, the defect(s) causing them is (are) normally located and corrected. Except for unanticipated hardware failures, these defects are the result of design or coding errors. By recording the kind of defects causing the failures, the software team accumulates information about the most probable errors in their design and coding practices. When it is possible to do so, modified practices can be adopted which prevent the insertion of those defects in the future.

Reported field failures can be recycles into testware design activities as well. In that case, your test design techniques are augmented to include the kinds of test cases that are sure to detect such defects in future code.

Declare this improvement wildly successful when the only software failures reported from the field are those due to requirement errors.

5. Improvement Strategies

5.1. A Cycle of Evolution

A process is "implemented" by providing those elements in the work environment that make it possible, by setting appropriate policies, and by convincing people that the following the given process is easier than not following it.

An implemented software process can be modeled in two layers which we call *architecture* and *implementation* in Figure 3, below. The architecture layer is a virtual framework, while the implementation layer is the actual physical embodiment of process capability. (The diagram is meant to suggest an IDEF0 node, with architecture playing the role of control and implementation playing the role of resource.) Notice the faint input/output arrows. They imply that the model does not convert inputs to outputs, but it does describe the capability for the process to deal with prescribed kinds of inputs and outputs.

The architecture layer contains

Activity Model. This is a description of an interrelated network of tasks to be performed for producing the process results. This description too often thought to be all there is to a process.

Techniques and Heuristics. These are descriptions of kinds of methods by which each of the tasks in the Activity Model may be performed.

Standards and Metrics. These are descriptions of the kinds of work products through which the tasks in the Activity Model are related.

The implementation layer provides the actual things (agents and materials) with which an occurrence of the process can happen.

Computing System, People. These are the agents of process activity, a collection of general purpose capabilities. Computing systems are generally thought to be more trainable and more reliable than human agents.

Tools. A tool is the usual way to embed a method and/or a standard into computing system agents.

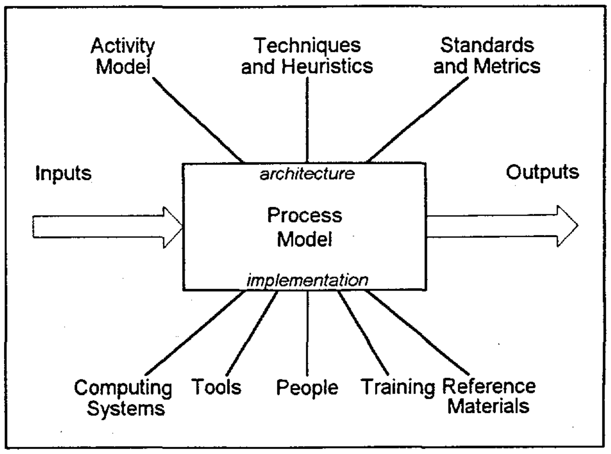

Figure 3. Elements of a Software Process Model

Tools implement the mechanics of various techniques and heuristics with which tasks are performed. Tools often provide *de facto* standards for the work product components they create.

Training. This is the usual way to embed methods and standards into human agents.

Reference Materials. These may be electronic or non-electronic recordings of various facts used by the agents during their activity. Document templates (providing document outline and style for a word processor and thematic explanations for the author) and standards documents are typical examples.

A process is "enacted" when it actually occurs. This is illustrated as the Project Application arrow in Figure 4. If the model includes the entire software development process, the process occurrence is called a "software project." The project converts actual instances of the prescribed inputs into instances of the outputs, hence the heavy input/output arrows.

Starting with the Software Process Model in the figure, a cycle of process evolution goes through these steps:

1. *One or more projects apply the model by realizing its implementation and "turning on the activity." (Application is more efficient by having implementation elements in reusable form.)*
2. *Measurements of the activities and work products are made during those projects. Some measurements are used as scaling factors (representing the project's actual inputs and outputs) while other measurements (after scaling) are attributed to the process model.*
3. *A separate software process improvement (SPI) project analyses the measurements and infers deficiencies in either the particular implementation elements of the process model, or in the architecture of the process model.*
4. *The process model architecture or the implementation is revised and made available for use in later projects. Figure 3 identifies those things which you may have to create or update so the process can be enacted again.*

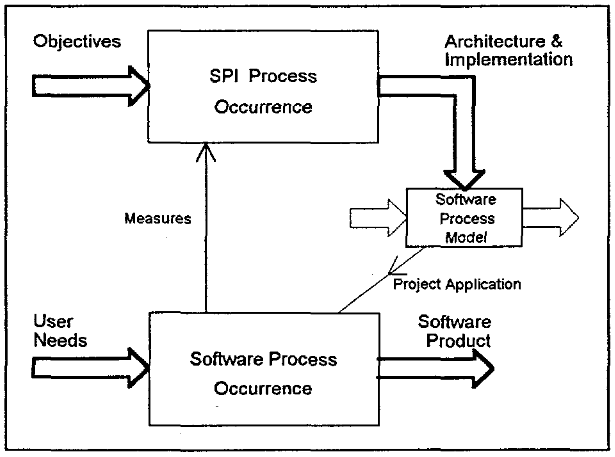

Figure 4. Cycle of Process Evolution

The goal of "continuous process improvement" is to keep this cycle active.

Notice that SPI is itself a process whose output is a process model. Thus, there is an SPI process model, too, though it is not shown in the figure. Resources should be available for supporting that process occurrence as well as for software projects.

5.2. Technical and Business Goals

Consider the testing process changes suggested in this paper. Which changes, if not all, would be most suitable for you? Which ones should you make first? Answering these questions requires you to understand the objectives that drive software process improvement in your organization. In this section, we describe some factors you should consider in developing your answers to these strategic questions. We consider objectives arising from technical goals and business goals. Conflicts among these must be resolved according to your unique circumstances.

Technical goals. These are the goals of the development staff for improving their work environment. Developers want to improve their work products, too, but often their first thoughts are to get enough "breathing space" just to think about what that would mean.

One strategy is this: First, establish the production of specific testing work products and later, improve the methods of producing them. By focusing first on the production of specific testing work products, developers and testers get visibility of their work and they don't have to recreate as much of it during the debugging cycles. Heuristics, templates, and tools should be formalized initially as prototypical elements of the process implementation, but they should be readily evolved as the easiest ways of dealing with the work products evolve in practice.

The order of improvements in Section 4 supports this strategy. The first two improvements are aimed at simply reducing the time required for doing your current quality of testing. The remaining ones then apply that saved time to increasing the quality of your testing.

In planning such improvements, it is invaluable to have some group, even *ad hoc*, to coordinate which improvements are being made and to share information about the improvements across the different projects. This is frequently called a Software Engineering Process Group (SEPG).[7]

Business goals. The strategy above may be too simplistic for you. Consider two common, yet rather different, drivers for testing process improvement.[8]

Reliability. This driver appears when you are being pressured to deliver fewer defects to customers. It is external in that it is usually raised directly by the customers. Defects are costing you and your customers too much in rework time, and are hurting the prospect of future sales.

Resources. This driver appears when you are being pressured to accomplish more with less. It is internal in that it probably isn't even visible to customers. You are looking for faster cycle time in development and maintenance, so you need more efficient testing.

If the first driver predominates, you may want to order the improvements to start with those which enable you to be more effective with the given level of effort.

If the second driver predominates, you may want to order the improvements so start with those which enable you to work faster with the same level of effectiveness you now have.

In either case, these are no longer technical issues. The proper evaluation and tradeoffs between drivers and impacts requires insight into a business' status and long-term objectives. These are managers' decisions, to be made by a Management Steering Group (MSG).[9] The MSG provides the vision to guide software process improvement, the leadership to pull success out of problems, and the resources to give it legitimacy.

This two-level scheme, with technical issues in the province of an SEPG and business issues in the province of a MSG, has been proven in practice. It is a suitable framework for implementing the software testing process changes suggested here.

Acknowledgments

The author thanks Dr. David Peercy of Sandia National Laboratories for his inexhaustible supply of independent critical thinking about software process improvement.

References

[1] For a general description of such a process, see Robert McFeely, "*IDEAL: A User's Guide for Software Process Improvement,*" Software Engineering Institute, CMU/SEI-HB-001, Feb. 1996. This is the successor to Robert S. McFeeley, David W. McKeehan, and Timothy Temple, "*Software Process Improvement Roadmap,*" Software Engineering Institute, Special Report CMU/SEI-94-SR-2, Mar. 1994.

[2] Testware includes test cases, a controlled test execution environment, agent(s) for setup, execution, and observation of the test cases, and test procedures.

[3] This summary of phase work products is general regarding software, but not systems. If the software is a component in a larger system, we assume a systems engineering framework in which functional requirements have been allocated to the software.

[4] Robert M. Poston, "Testing Tools Combine the Best of the New and Old," *IEEE Software*, vol. 12, no. 2, Mar. 1995, pp. 122-126.

[5] Brian Marick, "*The Craft of Software Testing,*" Prentice Hall, 1995.

6 Robert M. Poston and Mark W. Bruen, "Counting Down to Zero Software Failures," *IEEE Software*, vol. 4, no. 5, Sep. 1987, pp. 54-61.

[7] Priscilla Fowler and Stan. Rifkin, "*Software Engineering Process Group Guide,*" Software Engineering Institute, Technical Report CMU/SEI-90-TR-24, Sep. 1990.

[8] Robert V. Binder, "Design for Testability in Object-Oriented Systems," *Commun. ACM*, vol. 37, no. 9, Sep. 1994, pp. 87-101.

[9] See Reference 7.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.