# Unit Testing: Approaches, Tools, and Traps

**TOM ROOKER**
Abbott Analytical Products

Developing software is easy. It is just a matter of ones and zeros. The hard part is putting those ones and zeros in the proper order to perform useful work. Testing is one of the primary methods of demonstrating that the ones and zeros are in the proper order. Of the various testing levels unit testing is the most trumpeted and least understood. This article introduces a practical engineering approach to unit testing. The article answers the what, why, and how questions associated with unit testing. The approach is applicable to units destined for both software and firmware. The information presented is portable to any of the various programming languages. It is also scalable to meet the demands of complex projects. Finally, the presentation discusses the nonsilver bullet aspects of unit testing.

**Key words:** coverage analysis, framework, organizational focus, requirements traceability, risk analysis, test harness, test plan, unit test

**SQP References**

Design-Appropriate Test Cases
vol. 8, issue 1
Tom Griffin

Choosing a Tool to Automate Software Testing
vol. 2, issue 1
Mark Fewster and Dorothy Graham

## INTRODUCTION

Directors, managers, and software engineers often declare unit testing to be one of the foundational principles of software engineering. There are many views on how to execute this principle. This article brings together these views to suggest a common thread or approach. The article defines the scope of the term unit, explains the importance of unit testing, introduces the perspective of unit testing from various organizational environments, provides a method for taking a qualitative look at organizational focus, reviews current unit testing tactics, shows the utility of automated test frameworks, describes how to create unit test harness, highlights the value of a test plan, and discusses the tradeoffs associated with implementing unit testing. The approach presented in the article can be tailored to accommodate the software development environment of the software practitioner.

### Define the Unit

The acceptability of what a unit is depends on the user's perspective. A software system or program can be divided into one or more functional blocks or responsible entities called *units*. An equally valid definition of a unit is a software module or component of a design. The IEEE lists two definitions of a unit. First, it lists a unit as a separately testable element specified in the design of a computer software element. The second definition describes a unit as a logically separable part of a computer program. The IEEE also provides *component* and *module* as synonyms for unit (IEEE 1994). One recognized test advocate defines a unit as the smallest piece of software that can be independently tested (that is, compiled or assembled, loaded, and tested). That expert further states that a unit is usually the work of one programmer

consisting of a few hundred lines of source code (Beizer 1990). Another expert who specializes in object-oriented programming using extreme programming suggests that a class is equivalent to a unit (Beck 2000).

Regardless of how a user defines a unit, a system can be decomposed into one or more units. A typical decomposition is depicted in Figure 1. Note that unit 16 and unit 17 contain other units. This illustrates that a hierarchy of units is acceptable. A segment of code forms a unit. If this unit can be executed by itself using a driving mechanism and some number of supporting set-up structures or data, then it can undergo unit testing. Both the segment of code and any associated unit test entity become development artifacts.

## Why Unit Test?

The strategic rationale for performing unit testing is to provide a level of confidence that the segment of code will perform as expected. This rationale seeks to demonstrate that the segment of code meets the performance expectations of the customer as abstracted and implemented by the software practitioner. It fosters a positive, proactive approach to the software development process, and it couples tightly with segments of code destined for life critical and mission critical applications. This rationale strongly supports the tenets of extreme programming and agile programming, and is also valid for modification of segments of code found in large legacy systems.

To execute this strategy, unit testing tactics seek to identify bugs within a segment of code. Unit testing tactics generate a "find and fix" sequence of events to reduce the bug population of the code. The empirical equation of Table 1 reflects a strong likelihood that bugs will linger in untested code. Associated articles introduce the concept that there are subsets of bugs beyond the scope of the segment of code. These bugs may be transparent to testing or lie dormant until unforeseen conditions exist. This type of hidden bug is sometimes called *inherent* (Beatty 2000). This means that a unit test could be executed for an infinite amount of time and not find such a bug.

## Unit Test . . . When the Software Really Has to Work Every Time!

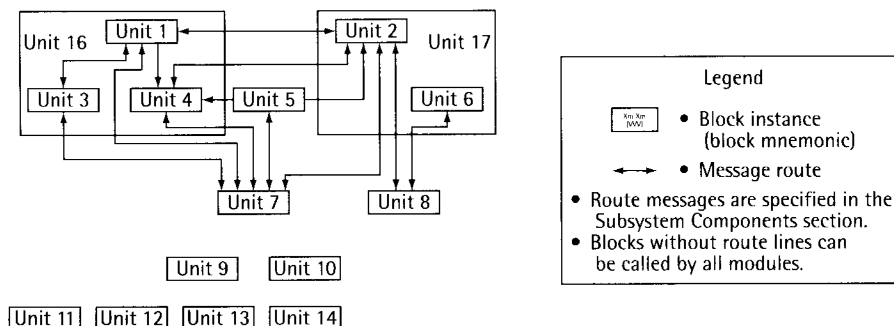
For life critical and mission critical applications segments of code must perform exactly as intended every time. These segments of code are typically built under rigorous specifications, such as FAA/RTCA do-178b, FAA/RTCA do-254, or FDA 820 QSR. Under such specifications the segments of code mature as the collection of artifacts grows. These rigorous specifications demand artifacts devoted to the unit test strategic

**TABLE 1** Paraphrased rules of thumb for untested code (Beizer 1990)

- For untested code: Residue bugs = Size of untested code * Probability of bugs
- High probability paths can be well tested at higher levels of testing.
- Paths with the highest probability of execution exhibit the fewest logic errors.
- The implementer's view on the probability of executing a path and its true probability are typically far apart.
- The implementer's perception of the importance of a code segment is strongly biased.

© 2006, ASQ

**FIGURE 1** System decomposition to 17 units




Legend
- Block instance (block mnemonic)
- Message route
- Route messages are specified in the Subsystem Components section.
- Blocks without route lines can be called by all modules.

© 2006, ASQ

rationale. The delivery of these artifacts becomes a gate for completing the application's project.

Several new software development models, such as extreme programming, agile programming, and Scrum, emphasize the strategic unit test rationale, demonstrating conformance to expectations. Although these development models may not be as structured or artifact oriented, a firm understanding of the expectations is manifested as the project matures. These development models encourage the generation of test cases even before code implementation begins. They also require that testing be done early and often. In this manner the software practitioner gains insight into the domain and the correct expected behaviors.

Modifications of code segments within large legacy systems create a need to demonstrate that the code continues to meet performance expectations. This need returns to the strategic rationale for unit testing. A legacy system might be composed of many millions of lines of code generated by thousands of software practitioners over many years and many releases. Adding a new feature to the legacy system adds a level of complexity and fog. If the segment of code implementing the new feature can be isolated from the legacy system, the complexity and fog can be reduced. Otherwise, the software practitioner will need to establish test cases suitable at the system level to make a judgment that the new segment of code conforms to expectations.

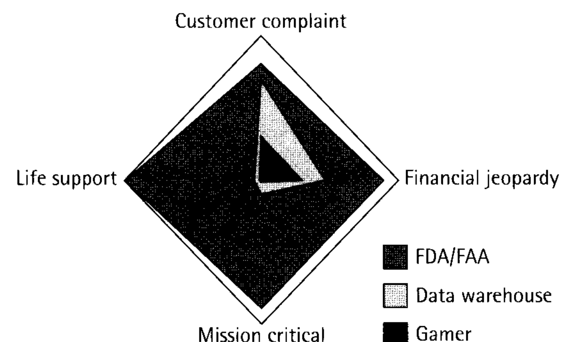
Organizational focus dictates when to unit test. The organizational focus can be schedule driven, requirements driven, document driven, quality driven, architecture driven, or technology driven (Booch 1995). The latest software development process models, such as those based on capability process maturity models and iterative incremental models, highlight the growing importance of unit testing (Sheard 1997). The level of process maturity and process standards does not dictate the exact approach to unit testing, but it does dictate the level of artifact development, review, and control.

The level of commitment to unit testing varies between software organizations and development environments. Organizations and environments express their level of commitment as expenditures of time, money, resources, and talent. The willingness to make the expenditures is driven by influences beyond the boundaries of the organization. A multispoke radar plot can be used to qualitatively measure the level of commitment of an organization or development environment. That commitment is qualitatively depicted as the shape formed by connecting the intersection points along the radar spokes. The resulting lines form a quadrilateral shaped area. Figure 2 employs the four spokes of customer complaints, financial jeopardy, mission critical, and life critical. When selling unit testing strategies to managers and software developers, the unit test advocate is justified in making additional spokes or substitutions. Such action would depend upon the organization or development environment focus.

The level of commitment in heavily regulated software organizations, such as those engaged in the FAA or FDA software product development, is very high. This is depicted in Figure 2 as a large quadrilateral shape that nearly fills the four-spoke radar plot. Figure 2 also shows a software company engaged in data warehouse applications. In this instance the associated commitment to unit testing is significantly less, as shown by the much smaller quadrilateral shape. This data warehouse software company is concerned about customer complaints and, to a lesser extent, financial jeopardy. Finally, a software company engaged in the production of video games has little interest in unit testing, as shown in Figure 2. The gamer firm is concerned about complaints and related financial losses. For the comparison presented in Figure 2, both the gamer and data warehouse firms are not applied in mission critical or life support roles.

Of the various development process models only the classic waterfall model has a single, rigorous defined unit test focal point or gate. This unit test gate occurs immediately after code implementation and before the start of integration testing, as shown in Table 2. The other commonly used development models incorporate

**FIGURE 2** Qualitative level of commitment to unit testing




© 2006, ASQ

unit testing activities into multiple sites. For example, extreme programming offers the unit test axiom of test early and test often. This axiom forces the use of unit test automation within the development process' build and release cycles.

# UNIT TEST TACTICS

Unit testing will normally combine a set of static and dynamic tactics. Static tasks such as requirements mapping/traceability, code review/walkthrough, and structural analysis do not directly run the code within the unit. Dynamic tasks, which include functional/black box test and coverage analysis/white box test, exercise the code within the unit. These tactics are briefly introduced in this section. Figure 3 depicts the branching of static and dynamic unit testing realms and basic elements.

## Static Unit Test Tactics

One of the most used static unit test tactics is requirements mapping/traceability. Requirements mapping/traceability starts with the systems requirements; requirements drive development and testing. They are the documentation of the expectations system behavior for a given set of conditions. Good requirements do not state how a system will work; instead, good requirements state what the system will do. It is possible to have both explicit and implied requirements. Requirements must be uniquely identified to unambiguously communicate behavior expectations. Each requirement has a potential risk or penalty associated with it. Understanding the system's risks or penalties provides a currency for making decisions governing the time and energy to be expended on a unit and its testing.

The power of requirements traceability is seen when requirements mapping is employed. Figure 4 shows a typical requirements mapping. This mapping shows that a single requirement can be mapped into more than one unit. It also displays a symbolic indication of the risk associated with each requirement. Each unit accumulates requirements and the highest risk indicator of its associated requirements. This provides a good idea of the spread of responsibilities among the units. In a similar manner the accumulation of a unit

**FIGURE 3** Unit test tactics branching into static and dynamic testing realms. Each testing realm is decomposed into elemental levels.

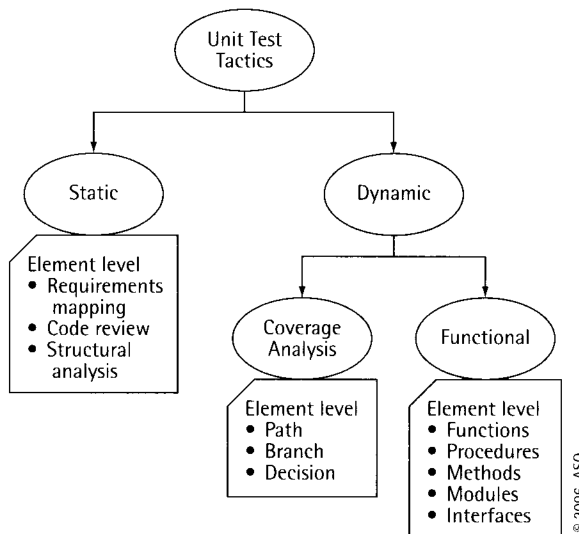



**TABLE 2** Development cycles

| Classical | Iterative (Booch 1995) | Rapid Early Prototype (Dicker 1993) | Extreme Programming (Waters 2000) |
|---|---|---|---|
| • Requirement | • Release1 | • Project plan | • User story |
| • Design | • Discovery | • Rapid analysis* | • Architectural spike |
| • Code Implementation | • Invention | • Database creation | • Planning game* |
| • Unit test* | • Implementation* | • Menus | • Spike |
| • Integration test | • Release2 | • Functions | • Iteration* |
| • System/Verification/ Validation test | • Discovery | • Prototype iteration* | • Functional testing* |
| | • Invention | • User approval | • Small releases |
| | • Implementation* | • Design derivation | |
| | • ReleaseN... | • Tuning | |
| | | • Maintenance | |

* Site of unit testing activities

© 2006, ASQ

subtotal ensures that at least one unit has responsibility for a specific requirement. Requirements mapping is the strategic step in requirements traceability. This requirements mapping/traceability will be carried to a lower level during unit test planning, which is discussed later in this article.

Figure 4 depicts a project requirement mapping using a spreadsheet approach. The mapping displays three requirements, BNC00005, BNC00006, and BNC00008, which were allocated as catastrophic according to the given risk analysis. Units 2, 3, 5, 8, 9, 10, 16, and 17 have responsibilities for supporting these three requirements. Unit 10 in particular seems to be carrying a substantial portion of the risk burden. This spreadsheet approach would substantiate placing first order of emphasis on the unit testing of unit 10. This mapping would suggest that units 2, 3, 5, 8, 9, 16, and 17 should receive second order emphasis. Units 1, 4, 6, 7, 11, 12, 13, 14, and 15 warrant a lesser order of emphasis. The expression *order of emphasis* may include management attention, time, resources, and so on, that a design project can bring to bear on its units and unit tests.

A less frequently used static unit test tactic is the code review/walkthrough. A code review/walkthrough normally involves at least the unit's programmer and a peer reviewer acting as a team. The reviewers seek to determine the unit's level of compliance with project coding standards. The review team checks for general code readability. The team ensures sufficient and correct commenting. The reviewers examine the unit for use constructs that could compromise functionality. This review can also provide a point at which to focus preliminary test plan and test cases preparations. Finally, a perception is developed on the readiness for testing.

The least-used static unit test tactic is structural analysis. Its use is restricted by the cost of owning the software tools to perform the structural analysis. The structural analysis involves measuring unit complexity. The value of complexity measure is limited since it is focused on unit structure as opposed to unit behavior. Complexity measures provide a metric on how efficiently a particular software development effort or group implements a design. Using complexity measuring tools, the code paths are mapped and indices are produced. These indices can be used to manage the complexity of the system by mandating the limits on complexity. Figure 5 illustrates unit dependencies. Figure 6 displays the expected code flow based upon a static analysis of the unit's code.

**FIGURE 4** Requirements mapping of 17 units

| Requirement: Note: Only the requirement number has been given for this example. | Risk analysis | Parent | Unit 1 | Unit 2 | Unit 3 | Unit 4 | Unit 5 | Unit 6 | Unit 7 | Unit 8 | Unit 9 | Unit 10 | Unit 11 | Unit 12 | Unit 13 | Unit 14 | Unit 15 | Unit 16 | Unit 17 | Interface | Hardware | Subtotal |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BNCM00001 | | 1 | | | | 1 | 1 | 1 | | | 1 | 1 | | | | | | 1 | | | | 7 |
| BNCM00002 | $E3 | 1 | 1 | 1 | | | | | | | 1 | 1 | | | | | | | 1 | | | 6 |
| BNCM00003 | | 1 | 1 | | | 1 | | | | 1 | 1 | 1 | | | | | | | | | 1 | 7 |
| BNCM00004 | | 1 | | 1 | 1 | | | | 1 | | | 1 | | | | | | | 1 | 1 | | 7 |
| BNCM00005 | !R5 | 1 | | 1 | | | | | | | | 1 | | | | | | 1 | 1 | | 1 | 6 |
| BNCM00006 | !R5 | 1 | | | | | 1 | | | | 1 | 1 | | | | | | 1 | | | | 5 |
| BNCM00007 | | 1 | | 1 | | | | | | | | 1 | | | | | | | | | 1 | 4 |
| BNCM00008 | !T6 | 1 | | | 1 | | | | | 1 | | 1 | | | | | | | | 1 | 1 | 6 |
| BNCM00009 | | 1 | 1 | | | | 1 | | | | | 1 | | | | | | 1 | | 1 | | 6 |
| BNCM000010 | +G2 | 1 | | | | | | | 1 | | | 1 | | | | | | | | 1 | | 4 |
| BNCM000011 | | 1 | | | | | | | | | | 1 | | | | | | | 1 | | 1 | 4 |
| BNCM000012 | +G2 | 1 | | | | | | | 1 | | 1 | | | 1 | | | | | | | | 4 |
| BNCM000013 | | 1 | 1 | | | | | | | | | | 1 | 1 | 1 | 1 | | | | 1 | | 7 |
| BNCM000014 | #T5 | 1 | | | | | 1 | | | | | | | | 1 | | | | 1 | | | 4 |
| BNCM000015 | | 1 | | | | | | | | | | 1 | | | | | | | | 1 | 1 | 4 |
| BNCM000016 | #Y5 | 1 | | | | | | | | | | | | | | | | | | | 1 | 2 |
| Subtotal | | 16 | 4 | 4 | 2 | 2 | 4 | 1 | 3 | 2 | 5 | 12 | 1 | 2 | 2 | 1 | 0 | 4 | 5 | 6 | 7 | |
| | | ! | $ | ! | ! | | ! | | + | ! | ! | ! | | + | # | | | ! | ! | + | ! | |

Risk Severity Category: !: Catastrophic. $: Critical. +: Major. #: Minor.

© 2006, ASQ

## Dynamic Unit Test Tactics

There are two types of dynamic unit testing. The first type is functional, or black box, testing. Functional testing accepts an external stimulus and yields an observable external response. The second type is white box testing or coverage analysis. This type of testing receives an external stimulus. The effect of this external stimulus is then observed as it propagates within the borders of the unit of interest.

Both functional and coverage analysis share the same high-level test flow, as shown in Figure 7. Both start by selecting a unit to test. Once the unit has been identified test cases can be designed for the unit. Given that there is more than one test case, the test cases will then be integrated into a sequence, which depends on project priorities. The test cases are then executed and the results evaluated. Several passes through this flow may be necessary until project test objectives can be fulfilled for the unit.

To facilitate cohesion, the following paragraphs have been labeled to refer to Figure 7. The paragraphs that focus on integration and running test cases are presented first. These paragraphs introduce tools for accomplishing the integrate and run tasks. The sequence of paragraphs that centers on the development of test cases follows, since they discuss the code-level preparation of units in anticipation of submittal to integration and run tasks. The paragraph devoted to unit test selection and evaluation forges the link to the previous tasks and is presented last.

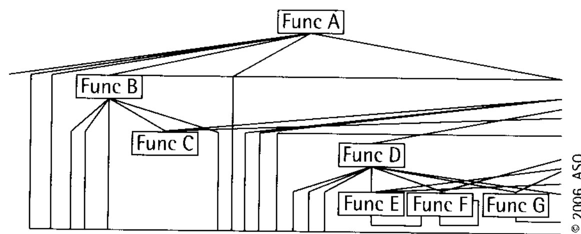
## Dynamic Unit Test Tool Alternatives: Focus on Integrate and Run

Although the following paragraphs introduce tools for integrating and running dynamic tests, it is possible to perform adequate unit testing with the same interpreters and compilers used to generate executable code. However, a software development environment that elects to take this approach will need a substantial level of skill and administration. The administration aspect not only includes the management of the test code but also of the expected and actual results. The advantage of taking this approach is that overhead associated with a tool, such as validation and learning curve, will not be experienced.

## Functional Unit Test: Focus on Integrate and Run

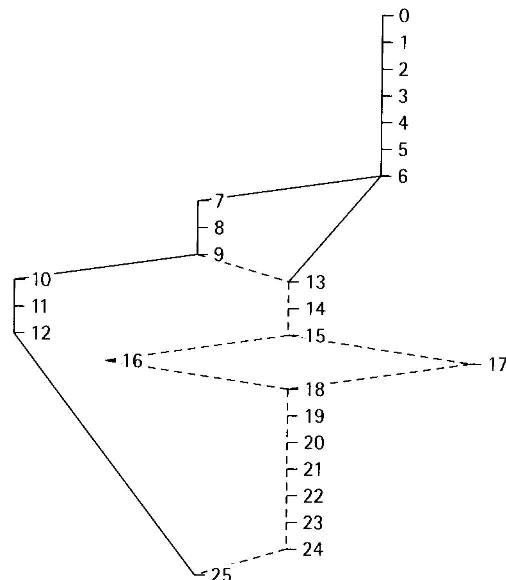
Functional unit testing has recently received endorsement from the extreme programming community. This boost has not only been in the form of test philosophy but also in the availability of tools. Although originally targeted at the extreme programming community, these tools can be applied to any software development environment. These tools include JUnit, CxxTest, and DUnit for Java, C++, and Delphi, respectively. (These tools, as well as others, can be obtained free of charge online at http://www.xprogramming.com/software.htm.) These tools have the advantage of being freeware, object oriented, related by a common framework, and on the leading edge of test technology and thought. But as with

**FIGURE 5** Functions mapping via McCabe showing functions




© 2006, ASQ

**FIGURE 6** Code paths via McCabe showing nodes




© 2006, ASQ

every upside there is a downside. The tools come with a variable level of documentation, and support is limited and varied. Also, there is no strict life-cycle control.

These tools have a common framework (Waters 2000). The framework is composed of a test case class with setUp and tearDown functions. This allows the user to isolate each test case from the others. The setUp and tearDown functions create and destroy the fixture or harness used in testing. Then if there is more than one test case, the user uses the framework's TestSuite class to add test cases using the addTestCase function and then the run function. This progresses into the TestResults class, where the error function handles anomalies beyond the test scope and the failure function manages anticipated events. Figure 8 displays a Delphi unit test with test cases, test suites, and metrics.

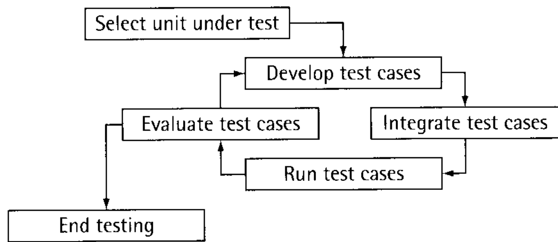
## Coverage Analysis: Focus on Integrate and Run

Unit test strategies employing coverage analysis use some combination of path, branch, and statement coverage. The likelihood of 100 percent test coverage of all paths, branches, and statements decreases as the complexity of a unit increases. The amount of time and testing resources available directly impacts the percentage of test coverage

Typically a test coverage analysis metric, C, is used to quantify this test activity. The coverage analysis metric can be developed by first letting Pp represent all possible control paths through a unit's code. Unit path testing verifies the control flow paths through the program by ensuring entry/exit points are reached. Path coverage can then be expressed as Cp=% of Pp. In a similar manner all possible decision branches within the unit can be represented by the term Pd. A decision or branch alternative has been tested if it is exercised at least once. Decision branch coverage can be expressed as Cd=% of Pd. Continuing with this development, all possible code statements within the unit can be expressed as Ps. Statement coverage attempts to execute as many code statements as possible at least once. Statement coverage can be expressed as Cs=% of Ps. Finally, the unit test coverage metric is the summation of the three coverage component percentages (C=Cp+Cd+Cs) (Beizer 1990). The test overage analysis metric, C, as well as its three components can each be adjusted to accommodate perceived or specified levels of risk within the unit being tested. With the test coverage metric, C, and its three components specified, the size and scope of the coverage testing activity can be planned, implemented, executed, and results measured.

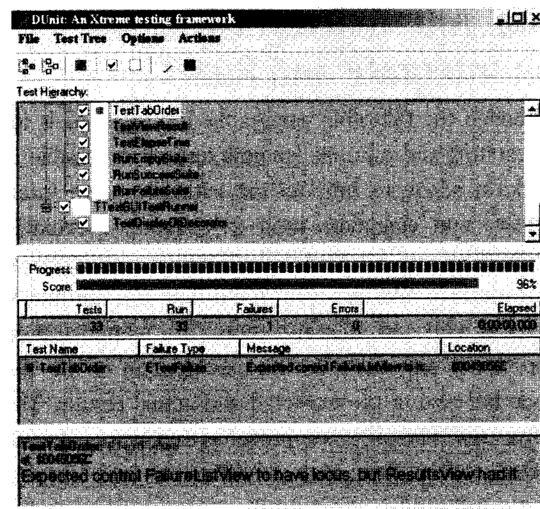
The coverage analysis implementation, execution, and results measurement hinge greatly on the testing tools used. Coverage analysis tools do not enjoy the same level of availability as the functional tools. However, the body freely available coverage analysis tool has steadily grown. An excellent example of a freely available test coverage tool is jcoverage for Java. A copy of jcoverage can be obtained online at http://www.xprogramming.com/software.htm. Jcoverage allows instrumentation down to line of code while still employing the advantages of JUnit for functional testing (see Figure 9). An additional feature is that the Java project can be tagged such that the source code is checked during each build to force compliance with a defined project coverage analysis standard. For such instances inadequate coverage testing is flagged as

**FIGURE 7** Dynamic unit test flow




© 2006, ASQ

**FIGURE 8** DUnit




© 2006, ASQ

a build failure. In addition to having early release aspects, jcoverage has the same issues as JUnit of documentation, support, and life-cycle control.

## Unit Test Harnesses: Focus on Develop

Functional unit testing and coverage analysis concepts, tools, and frameworks are useless unless the unit of code can be harnessed for testing. The actual construction of a unit's test harness can be started once the unit allocations have been made. The extreme programming model advocates this early involvement. More structured development, such as the classical waterfall, delays creation of the unit test harness until suitable project maturity gates have been passed. To actually build a test harness project/top-level and test code/low-level decisions need to be made.

The first project-level decision that must be made is whether the field code will contain the embedded unit test code harnessing or whether a separate version of the source code will retain the unit test harness code. Another top-level decision that must be made concerns whether functional and coverage analysis testing can be merged into one harness or whether separate harnesses will be required. The final project-level decision that needs to be addressed is the amount of regression testing necessary against the original source code to guarantee the same results and behavior. Once these three decisions have been made, the unit test harness can be constructed.

At the test code/low level there is one foundational decision that must be made concerning the construction of the test harness. Unit test harnesses can be constructed using the manual hard-coding instrumentation approach or the state machine tool approach.

## Hard-Code Approach to Test Harness Generation: Focus on Develop

The manual hard-code approach is brute force instrumentation of the source code. The instrumentation is used to provide an execution trail of the unit. If conditional compile syntax features of the programming language are used, the test instrumentation can be embedded within the source code without conveying it to the executable version of code. Figure 10 displays a hard-coded technique designed to demonstrate rudimentary path and statement coverage involving a function called runjackpot(). At its most rudimentary, the individual designing the test must first embed instrumentation in the form of calls to a supporting print function to record the passing of a specific point in the source code. The next step is to collect a set of expected results for a run through a single pass of this portion of code. Now if an independent tester executes the instrumented runjackpot() function, an observed set of test results is obtained. If the observed and expected results compare favorably, then one can conclude that the test passed.

**FIGURE 9** Source code showing jcoverage logger instrumentation

```
public int square(int x) {
    if (logger.isDebugEnable()) {
        logger.debug( x:  +x);
    }
    int result=x*x;
    if (logger.isDebugEnabled()) {
        logger.debug( result:  +result);
    }
    return result;
}
```

© 2006, ASQ

**FIGURE 10** Primitive hard-coded instrumentation of source code

```
int runjackpot()
{
    fprintf( utmfile, "runjackpot() begining \n");
... Processing Steps ...
    fprintf( utmfile, "runjackpot() going to Start ()\n");
        Start();
    fprintf( utmfile, "runjackpot() leaving to test driver \n");
    return 0;
}
```

© 2006, ASQ

The major advantage of the rudimentary manual hard-code approach is that in heavily regulated development environments there is no need to validate the hard-coding tool, since it is merely using the available language syntax to achieve a test objective or requirement. A more elegant employment of the manual hard-code approach leverages the benefit that is to be gained from frameworks and test libraries. In this instance the test developer builds the unit test harness instrumentation using the syntax and semantics of a test framework such as CxxTest or jcoverage. The problem with the more elegant approach is that framework may need to be validated before use in a heavily regulated development environment.

Both the rudimentary and the more elegant manual hard-code approaches have two major disadvantages. The first disadvantage is that they rely heavily on the skill set of the test developer. The second disadvantage is that test harness development process is not highly repeatable or reusable.

## Lex-Yacc State Machine Test Harness Generator: Focus on Develop

The lex-yacc state machine approach overcomes the major disadvantages of the hard-code approach. (Lex and yacc are software tools commonly associated with transforming structured inputs into an alternate format. The most common use of lex and yacc is as a compiler.) The lex-yacc state machine approach is highly repeatable and reusable. The repeatability and reusability are derived from embedding the target unit's programming language syntax rules and test harness function calls and test statements within a state machine. Using the embedded syntax rules the lex-yacc state machine synthesizes a test version of the original unit. There are readily available freeware lex-yacc packages such as tply41a.zip (Graef 2000) that provide suitable lexicon analyzer and parser capability. More recently, lex-yacc state machines have been integrated with graphical user interfaces. This integration allows the test developer to automatically generate a ready-to-run test version of a unit while masking the lex-yacc state machine complexity. Two such test generation programs are available online free of charge from http://qed1.home.mindspring.com. Although the resulting test harness is ready to run, the test developer may still need to provide sufficient engineering to ensure that the test version incorporates sufficient test cases to fulfill testing objectives and requirements. This is accomplished using the ready-to-run version as a template to replicate additional test cases in iterative sequences of cut, paste, and modify. With the modifications completed, the unit test harness is ready to run under appropriate test frameworks, such as CxxTest or JUnit.

As mentioned previously, the lex-yacc state machine approach has the advantages of repeatability and reusability; however, it has two major disadvantages. First, it is a tool that would require an additional validation effort if used in a rigorous FDA or FAA software development environment. Second, the user must rely on the skill set of at least one layer of test tool builders. A second order disadvantage is that this approach still depends to a lesser extent on the skill set of the unit test developer to incorporate sufficient test cases to achieve test objectives and requirements.

## How to Get It Done: Focus on Select and Evaluate

The process for finally "getting it done" depends upon the organizational focus. Highly regulated software development environments may wrap their unit testing in structured test artifacts, risks analysis, validated test cases, and witnessed test runs. Less-regulated development environments may rely upon unit testing embedded in automated builds, which yield e-mail notifications of anomalies. Underpinning both of these extremes is the unit test plan. The plan may be a very formal artifact for a highly regulated development environment. Alternately, for the less regulated development environment it may be a set of scripted thoughts and ideas. In both cases the unit test plan will have a description, risk analysis, mix of unit test tactics, test setup/harness definitions, test cases, test run documentation, and test results. Both unit-testing extremes are valid for their respective organizational focus. Both unit-testing extremes can make significant contributions to the effort of removing defects from units.

## Traps

The first unit test trap is the silver bullet or the expectation that unit testing will find all defects. Unit testing is both effective and ineffective at finding bugs. Table 3

**TABLE 3** Unit test bug finding (Beatty 2000)

| Code review/walkthrough | Functional testing | Coverage analysis |
|---|---|---|
| • Effective against:<br>- Math overflow/underflow<br>- Interrupt handling<br>• Ineffective against:<br>- Nonimplementation errors<br>- Version control errors<br>- Resource sharing problems<br>- Interface | • Effective against:<br>- Resource mapping<br>- Instrumentation problem<br>- Interface<br>• Ineffective against:<br>- Math overflow/underflow<br>- Reentrance problem<br>- Improper variable initialization<br>- Interrupt handling/ Interrupt suppression<br>- Memory allocation/de-allocation | • Effective against:<br>- Off by "1"<br>- Parameter passing<br>- Logic/math processing error<br>- Incorrect control flow<br>• Ineffective against:<br>- Data synchronization error<br>- Interrupt handling/ Interrupt suppression<br>- Task synchronization<br>- Stack overflow/underflow<br>- Resource mapping<br>- Instrumentation problem |

© 2006, ASQ

shows the bug types against which unit testing is very effective and against which it is ineffective. Based on Table 3 it is possible to state that even with unit testing there are still holes that must be plugged by some other means.

Coupled with the silver bullet trap is the trap resulting from the failure to balance the cost and benefits of unit testing. The next trap is the failure to look at the defect pathology driving unit testing. When an individual fails to investigate a defect that arises from unit testing, that person is failing to identify the root cause of the issue or the defect pathology. The final trap results from the suboptimization view replacing a system view. The term *suboptimization* implies that the parts begin to grow more important than the whole. Unit test suboptimization begins to grow as a crucial unit begins to overtake the importance and resources at the expense of the system and is other units.

# CONCLUSION

There are several points that can be made in conclusion. First, unit testing is not for the faint of heart. It takes substantial effort at a number of organizational levels. To be effective, unit testing must have an organized and balanced plan. The unit testing plan can be tailored to conform to the risks, company focus, and engineering decisions. There are valid engineering approaches to unit testing. The results will vary according to company focus, project resources, and skill set of participants.

## REFERENCES


Beatty, S. 2000. Sensible software testing. *Embedded Systems Programming* (August): 98-121.

Beck, K. 2000. Simple smalltalk testing: With patterns. Available at http://www.xprogramming.com/testfram.htm .

Beizer, B. 1990. *Software testing techniques, 2nd edition.* New York: International Thomson Computer Press.

Booch, G. 1995. *Object solutions: Managing the object-oriented project.* Menlo Park: Pearson Education.

Dicker, C. 1993. Is it designed right? *Unix Review* (June): 51-58.

Graef, A. 2000. Turbo Pascal Lex/Yacc. Available at http://www.musikwissenschaft.uni-mainz.de/~ag/tply/tply.html

IEEE. 1994. IEEE Std. 100-1994. Available at http://www.fda.gov/ora/inspect_ref/igs/gloss.html .

Sheard, S. A. 1997. The frameworks quagmire: A brief look. *Crosstalk* (September).

Waters, J. 2000. Extreme method simplifies development puzzle. *Application Development Trends* (July): 20-26.


## BIOGRAPHY


**Tom Rooker** is a contractor engaged in hardware/software reliability for avionics. Prior contract assignments include software projects for point-of-sale transactions, Web-based commerce, and medical communication devices. Rooker has played a pivotal role in the transfer of knowledge within the regional software and reliability engineering community. His specific interests are software engineering process and reliability engineering tools. He holds the copyrights to a suite of software and reliability engineering tools.

Rooker was previously employed at SAS, where he was responsible for the development of automated unit testing and Web component testing tools. Prior to that he provided reliability engineering at Telex and Texas Instruments supporting computer communication products, large-scale integrated circuits, and laser-guided weapons. He is a licensed professional engineer and a certified reliability engineer. He can be reached by e-mail at qed1@mindspring.com .