

What Is Software Testing? And Why Is It So Hard?

Software testing is arguably the least understood part of the development process. Through a four-phase approach, the author shows why eliminating bugs is tricky and why testing is a constant trade-off.

James A. Whittaker, *Florida Institute of Technology*

Virtually all developers know the frustration of having software bugs reported by users. When this happens, developers inevitably ask: How did those bugs escape testing? Countless hours doubtless went into the careful testing of hundreds or thousands of variables and code statements, so how could a bug have eluded such vigilance? The answer requires, first, a closer look at software testing within the context of development. Second, it requires an understanding of the role

software testers and developers—two very different functions—play.

Assuming that the bugs users report occur in a software product that really is in error, the answer could be any of these:

- *The user executed untested code.* Because of time constraints, it's not uncommon for developers to release untested code—code in which users can stumble across bugs.
- *The order in which statements were executed in actual use differed from that during testing.* This order can determine whether software works or fails.
- *The user applied a combination of untested input values.* The possible input combinations that thousands of users can make across a given software interface are simply too numerous for testers to apply them all. Testers must make tough decisions about which inputs to

test, and sometimes we make the wrong decisions.

- *The user's operating environment was never tested.* We might have known about the environment but had no time to test it. Perhaps we did not (or could not) replicate the user's combination of hardware, peripherals, operating system, and applications in our testing lab. For example, although companies that write networking software are unlikely to create a thousand-node network in their testing lab, users can—and do—create such networks.

Through an overview of the software testing problem and process, this article investigates the problems that testers face and identifies the technical issues that any solution must address. I also survey existing classes of solutions used in practice.

Readers interested in further study will find the sidebar “Testing Resources” helpful.

Testers and the Testing Process

To plan and execute tests, software testers must consider the software and the function it computes, the inputs and how they can be combined, and the environment in which the software will eventually operate. This difficult, time-consuming process requires technical sophistication and proper planning. Testers must not only have good development skills—testing often requires a great deal of coding—but also be knowledgeable in formal languages, graph theory, and algorithms. Indeed, creative testers have brought many related computing disciplines to bear on testing problems, often with impressive results.

Even simple software presents testers with obstacles, as the sidebar “A Sample Software Testing Problem” shows. To get a clearer view of some of software testing’s inherent difficulties, we can approach testing in four phases:

- Modeling the software’s environment
- Selecting test scenarios

- Running and evaluating test scenarios
- Measuring testing progress

These phases offer testers a structure in which to group related problems that they must solve before moving on to the next phase.

Every software development organization tests its products, yet delivered software always contains residual defects of varying severity. Sometimes it’s hard to imagine how a tester missed a particularly glaring fault. In too many organizations, testers are ill-equipped for the difficult task of testing ever-more-complex software products. Informal surveys of seminar audiences suggest that few of those who perform testing (either as a profession or as an adjunct to development or other roles) have been adequately trained in testing or have software testing books on their desks.

James Whittaker sheds some light on why testing today’s software products is so challenging, and he identifies several solid approaches that all testers should be able to thoughtfully apply. The effective tester has a rich toolkit of fundamental testing techniques, understands how the product will be used in its operating environment, has a nose for where subtle bugs might lurk in the product, and employs a bag of tricks to flush them out. The methods described here can help testers provide a sensible answer to the question of what they really mean when they say they are done testing a software system.

—Karl Wiegers and Dave Card, *Nuts & Bolts* editors

Testing Resources

Literature on software testing has appeared since the beginning of computer science but the paper that defined the field and instigated much of today’s research agenda is John Goodenough and Susan Gerhart’s classic, “Toward a Theory of Test Data Selection” (*IEEE Trans. Software Eng.*, June 1975). That same year began the prolific research career of William Howden, whose papers and book chapters have helped shape the field (see www-cse.ucsd.edu/users/howden for a complete listing). A complete list of testing researchers along with links to their home pages can be found at Roland Untch’s Storm site (www.mtsu.edu/~storm).

The seminal book on testing was Glenford Myers’ *The Art of Software Testing* (John Wiley & Sons, 1979), which was the only testing book of note for years. Today, however, a search of amazon.com for “software testing” yields more than 100 matches. Among those books, a handful are considered contenders to succeed Myers’.

Brian Marick’s *The Craft of Software Testing* (Prentice Hall, 1995) is my pick for a solid introduction to the subject and is full of good advice for handling tough testing problems. In addition, the author is active at posting updates to the appen-

dices and interacting with his readers on his Web site (www.rst-corp.com/marick). Cem Kaner’s *Testing Computer Software* (The Coriolis Group, 1993) is very popular among industry practitioners for its easy reading and good examples. The best-selling testing book is standard issue for new testers at many of the best software testing companies. Boris Beizer’s *Black Box Testing* (John Wiley & Sons, 1995) is chock-full of examples and is perhaps the most methodical and prescriptive of the testing books. It also provides a good treatment of using graph techniques to test programs. Finally, I like Bill Hetzel’s *The Complete Guide to Software Testing* (John Wiley & Sons, 1993) for its thoroughness. It is one of the few testing books that discusses testing process and life-cycle activity.

If you prefer finding free information on the Web, I suggest visiting the Storm site mentioned earlier and also highly recommend Bret Pettichord’s software testing hotlist: an annotated list of links to some of the Web’s best testing information (www.io.com/~wazmo/qa). Finally, you might consider participating in Danny Faught’s discussion list (swtest-discuss@rsn.hp.com) or the newsgroup comp.software.testing.

A Sample Software Testing Problem

A small program displays a window with the current system time and date, which can be changed by typing new values into the edit fields as shown in Figure A. The program is terminated by the Alt-F4 keystroke sequence, and the Tab key moves between fields.

In deciding how to test this (or any) program, a software tester considers the environment in which the software operates; the source code that defines the software; and the interface between the software and its environment.

Environment

Software exists in an environment in which other entities (users) stimulate it with inputs. The software provides those users with output. This example program has two input sources, the obvious human user who supplies inputs from the set {time, date, Tab, Alt-F4}, and the operating system “user” that supplies memory for the program to run and supplies the current system time and date as an application service.

A diligent tester will consider the valid inputs from each of these sources as well as invalid and unexpected inputs. What if the human user types other Alt-sequences or keystrokes outside the acceptable input set? What if available memory is insufficient for the program to run? What if the system clock malfunctions? Testers must consider these possibilities, select the most important ones, and figure out how to simulate these conditions.

Testers next think about how users interact in ways that might cause the software to fail. What happens, for example, when some other program changes the time and date—does our application properly reflect this change? Today’s multitasking operating systems demand that testers think through such scenarios.

Source code

The code for this application might have a While loop similar to the one in Figure B.

How many test cases does it take to fully cover, or exercise, the source code? To determine this, we evaluate each con-

| | |
|---------------------------|-----------------|
| Current Time: 9:28:32pm | New Time: _____ |
| Current Date: 24 Aug 1999 | New Date: _____ |

Figure A. Current system time and date, along with fields for entering new values.

dition to both true and false by means of a truth table. We thus execute not only each source statement, but we also cover each possible branch in the software. The truth table in Figure C documents each possible combination of conditions in the While loop, the three parts of the Case statement, and the nested If statements.

```
Input = GetInput()
While (Input ≠ Alt-F4) do
  Case (Input = Time)
    If ValidHour(Time.Hour) and ValidMin(Time.Minute) and
      ValidSec(Time.Second) and ValidAP(Time.AmPm)
    Then
      UpdateSystemTime(Time)
    Else
      DisplayError("Invalid Time.")
    Endif
  Case (Input = Date)
    If ValidDay(Date.Day) and ValidMnth(Date.Month) and
      ValidYear(Date.Year)
    Then
      UpdateSystemDate(Date)
    Else
      DisplayError("Invalid Date.")
    Endif
  Case (Input = Tab)
    If TabLocation = 1
    Then
      MoveCursor(2)
      TabLocation = 2
    Else
      MoveCursor(1)
      TabLocation = 1
    Endif
  Endcase
  Input = GetInput()
Enddo
```

Figure B. Sample source code demonstrating a While loop.

Phase 1: Modeling the Software’s Environment

A tester’s task is to simulate interaction between software and its environment.

Testers must identify and simulate the interfaces that a software system uses and enumerate the inputs that can cross each interface. This might be the most fundamental

These eight possible cases cover only statements and branches. When we consider how each complex condition in the If statements actually gets evaluated, we must add several more cases. Although there is only one way for these statements to

| Possible cases | While | Case 1 | If 1 | Case 2 | If 2 | Case 3 | If 3 |
|----------------|-------|--------|------|--------|------|--------|------|
| 1 | F | — | — | — | — | — | — |
| 2 | T | T | T | — | — | — | — |
| 3 | T | T | F | — | — | — | — |
| 4 | T | F | — | T | T | — | — |
| 5 | T | F | — | T | F | — | — |
| 6 | T | F | — | F | — | T | T |
| 7 | T | F | — | F | — | T | F |
| 8 | T | F | — | F | — | F | — |

Figure C. A truth table such as this helps keep track of all possible combinations of inputs.

evaluate true (that is, every condition must be true for the statement to be true), there is more than one way for the first two If statements to evaluate false. In fact, we'd find that there are $2^x - 1$ ways (where x is the number of conditions in the statement).

Using this logic, there are $2^4 - 1 = 15$ ways to execute the third test and $2^3 - 1 = 7$ ways to execute the fifth (each of these cases appears in bold, above), for a total of 28 test cases. Now, imagine how many test cases would be required to test a software system with a few hundred thousand lines of code and thousands of such complex conditions to evaluate. It's easy to see why software is commonly released with unexecuted source code.

In addition to covering the source code, testers also must think about missing code. The fact that the Case statement has no default case could present problems.

Interface

Besides testing the environment and the source code, we must also determine the values assigned to the specific data that crosses the interface from the environment to the software under test; for example, *Time* and *Date*. Variable input is difficult to test because many variable types can assume a wide range of possible values. How many different times are there in a day? The combinatorics aren't encouraging: 12 hours \times 60 minutes \times 60 seconds \times 2 am/pm for a total of 86,400 different input values. That's just the valid values; invalid values like 29 o'clock must also be tested.

Next we must consider the possible legal and illegal values for the date field, and finally, decide on specific combinations of time and date to enter simultaneously—like midnight of the year 1999. This is enough to overwhelm even the biggest testing budget.

Finally, we must determine which inputs will be applied consecutively during testing. This is, perhaps, the most subtle and elusive aspect of testing. Obviously, the first input to be applied is the one that invokes the software. Next, we must choose to apply one of the other inputs, choose another to follow that, and so on until we exit the software. Much can happen during such sequencing. Will the software accept several consecutive Tab keys? Will it handle a change to the Time field only (leaving the Date field unchanged), the Date field only, and also changes to both? The only way to find out is to apply each of these cases separately.

How many cases are there? Since the While loop is unbounded, there is no upper limit. Testers have two ways to handle infinite input domains. First, we might isolate infinite input subsets into separate subdomains,¹ decomposing the problem into smaller problems.

Second, as in development, we can abstract; here, inputs into events. Rather than deal with specific physical inputs such as mouse clicks and keystrokes, testers create abstract events that encompass a number of physical input sequences. We did this in the example above by creating the inputs *Time* and *Date*. During analysis of the input domain, testers can use these abstractions to think through the problem. When the test scenario is actually implemented, testers can replace the abstraction with one of its possible physical instantiations. (I use "scenario" to mean simply "instructions about what things to test." A more precise term is "test case," which implies exact specification of initial conditions, inputs to apply, and expected outputs.)

Reference

1. E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Trans. Software Eng.*, Vol. 6, No. 3, May 1980, pp. 236–246.

issue that testers face, and it can be difficult, considering the various file formats, communication protocols, and third-party (application programming interfaces) available. Four

common interfaces are as follows:

- *Human interfaces* include all common methods for people to communicate with soft-

If code and input coverage were sufficient, released products would have very few bugs.

ware. Most prominent is the GUI but older designs like the command line interface and the menu-driven interface are still in use. Possible input mechanisms to consider are mouse clicks, keyboard events, and input from other devices. Testers then decide how to organize this data to understand how to assemble it into an effective test.

- *Software interfaces*, called APIs, are how software uses an operating system, database, or runtime library. The services these applications provide are modeled as test inputs. The challenge for testers is to check not only the expected but also the unexpected services. For example, all developers expect the operating system to save files for them. The service that they neglect is the operating system's informing them that the storage medium is full. Even error messages must be tested.
- *File system interfaces* exist whenever software reads or writes data to external files. Developers must write lots of error-checking code to determine if the file contains appropriate data and formatting. Thus, testers must build or generate files with content that is both legal and illegal, and files that contain a variety of text and formatting.
- *Communication interfaces* allow direct access to physical devices (such as device drivers, controllers, and other embedded systems) and require a communication protocol. To test such software, testers must be able to generate both valid and invalid protocol streams. Testers must assemble—and submit to the software under test—many different combinations of commands and data, in the proper packet format.

Next, testers must understand the user interaction that falls outside the control of the software under test, since the consequences can be serious if the software is not prepared. Examples of situations testers should address are as follows:

- Using the operating system, one user deletes a file that another user has open. What will happen the next time the software tries to access that file?
- A device gets rebooted in the middle of a stream of communication. Will the software realize this and react properly or just hang?
- Two software systems compete for duplicate services from an API. Will the API correctly service both?

Each application's unique environment can result in a significant number of user interactions to test.

Considerations

When an interface presents problems of infinite size or complexity, testers face two difficulties: They must carefully select values for any variable input, and they must decide how to sequence inputs. In selecting values, testers determine the values of individual variables and assign interesting value combinations when a program accepts multiple variables as input.

Testers most often use the *boundary value partitioning* technique¹ for selecting single values for variables at or around boundaries. For example, testing the minimum, maximum, and zero values for a signed integer is a commonly accepted idea as well as values surrounding each of these partitions—for example, 1 and -1 (which surround the zero boundary). The values between boundaries are treated as the same number; whether we use 16 or 16,000 makes no difference to the software under test.

A more complex issue is choosing values for multiple variables processed simultaneously that could potentially affect each other. Testers must consider the entire cross product of value combinations. For two integers, we consider both positive, both negative, one positive and one zero, and so forth.²

In deciding how to sequence inputs, testers have a sequence generation problem. Testers treat each physical input and abstract event as symbols in the alphabet of a formal language and define a model of that language. A model lets testers visualize the set of possible tests to see how each test fits the big picture. The most common model is a graph or state diagram, although many variations exist. Other popular models include regular expressions and grammars, tools from language theory. Less-used models are stochastic processes and genetic algorithms. The model is a representation that describes how input and event symbols are combined to make syntactically valid words and sentences.

These sentences are sequences of inputs that can be applied to the software under test. For example, consider the input `Filemenu.Open`, which invokes a file selection dialog box; `filename`, which represents the selection (with mouse clicks, perhaps) of an existing file, and `ClickOpen` and `ClickCancel`,

which represent button presses. The sequence `Filemenu.Open filename ClickOpen` is legal, as are many others. The sequence `ClickCancel Filemenu.Open` is impossible because the cancel button cannot be pressed until the dialog box has been invoked. The model of the formal language can make such a distinction between sequences.

Text editor example

We can represent legal uses of the file selection dialog in, for example, a text editor with the regular expression:

```
Filemenu.Open filename* (ClickOpen | ClickCancel)
```

in which the asterisk represents the Kleene closure operator indicating that the `filename` action can occur zero or more times. This expression indicates that the first input received is `Filemenu.Open` followed by zero or more selections of a filename (with a combination of mouse clicks and keyboard entries), then either the Open or Cancel button is pressed. This simple model represents every combination of inputs that can happen, whether they make sense or not.

To fully model the software environment for the entire text editor, we would need to represent sequences for the user interface and the operating system interface. Furthermore, we would need a description of legal and corrupt files to fully investigate file system interaction. Such a formidable task would require the liberal use of decomposition and abstraction.

Phase 2: Selecting Test Scenarios

Many domain models and variable partitions represent an infinite number of test scenarios, each of which costs time and money. Only a subset can be applied in any realistic software development schedule, so how does a smart tester choose? Is 17 a better integer than 34? How many times should a filename be selected before pressing the Open button?

These questions, which have many answers, are being actively researched. Testers, however, prefer an answer that relates to coverage of source code or its input domain. Testers strive for *coverage*: covering code statements (executing each source line at least once) and covering inputs (applying each externally generated event). These are the

minimum criteria that testers use to judge the completeness of their work; therefore, the test set that many testers choose is the one that meets their coverage goals.

But if code and input coverage were sufficient, released products would have very few bugs. Concerning the code, it isn't individual code statements that interest testers but *execution paths*: sequences of code statements representing an execution of the software. Unfortunately, there are an infinite number of paths. Concerning the input domain, it isn't the individual inputs that interest testers but *input sequences* that, taken as a whole, represent scenarios to which the software must respond. There are an infinite number of these, too.

Testers sort through these infinite sets to arrive at the best possible *test data adequacy criteria*, which are meant to adequately and economically represent any of the infinite sets. "Best" and "adequately" are subjective; testers typically seek the set that will find the most bugs. (High and low bug counts, and their interpretation, are discussed later). Many users and quality assurance professionals are interested in having testers evaluate *typical use* scenarios—things that will occur most often in the field. Such testing ensures that the software works as specified and that the most frequently occurring bugs will have been detected.

For example, consider the text editor example again. To test typical use, we would focus on editing and formatting since that is what real users do most. However, to find bugs, a more likely place to look is in the harder-to-code features like figure drawing and table editing.

Execution path test criteria

Test data adequacy criteria concentrate on either execution path coverage or input sequence coverage but rarely both. The most common execution path selection criteria focus on paths that cover control structures. For example,

- Select a set of tests that cause each source statement to be executed at least once.
- Select a set of tests that cause each branching structure (If, Case, While, and so on) to be evaluated with each of its possible values.

However, control flow is only one aspect of the source code. What software actually

"Best" and "adequately" are subjective; testers typically seek the set that will find the most bugs.

How much retesting of version n is necessary using the tests that were run against version $n - 1$?

does is move data from one location to another. The *dataflow* family of test data adequacy criteria³ describe coverage of this data. For example,

- Select a set of tests that cause each data structure to be initialized and then subsequently used.

Finally, *fault seeding*, which claims more attention from researchers than practitioners, is interesting.¹ In this method, errors are intentionally inserted (seeded) into the source code. Test scenarios are then designed to find those errors. Ideally, by finding seeded errors, the tester will also find real errors. Thus, a criterion like the following is possible:

- Select a set of tests that expose each of the seeded faults.

Input domain test criteria

Criteria for input domain coverage range from simple coverage of an interface to more complex statistical measurement.

- Select a set of tests that contain each physical input.
- Select a set of tests that cause each interface control (window, menu, button, and so on) to be stimulated.

The *discrimination* criterion⁴ requires random selection of input sequences until they statistically represent the entire infinite input domain.

- Select a set of tests that have the same statistical properties as the entire input domain.
- Select a set of paths that are likely to be executed by a typical user.

Summary

Testing researchers are actively studying algorithms to select minimal test sets that satisfy criteria for execution paths and input domains. Most researchers would agree that it is prudent to use multiple criteria when making important release decisions. Experiments comparing test data adequacy criteria are needed, as are new criteria. However, for the present, testers should be aware which criteria are built into their methodology and understand the inherent

limitations of these criteria when they report results.

We'll revisit test data adequacy criteria in the fourth phase, test measurement, because the criteria also serve as measures of test completeness.

Phase 3: Running and Evaluating Test Scenarios

Having identified suitable tests, testers convert them to executable form, often as code, so that the resulting test scenarios simulate typical user action. Because manually applying test scenarios is labor-intensive and error-prone, testers try to automate the test scenarios as much as possible. In many environments, automated application of inputs through code that simulates users is possible, and tools are available to help.

Complete automation requires simulation of each input source and output destination of the entire operational environment. Testers often include data-gathering code in the simulated environment as test hooks or asserts. This code provides information about internal variables, object properties, and so forth. These hooks are removed when the software is released, but during test scenario execution they provide valuable information that helps testers identify failures and isolate faults.

Scenario evaluation, the second part of this phase, is easily stated but difficult to do (much less automate). Evaluation involves the comparison of the software's actual output, resulting from test scenario execution, to its expected output as documented by a specification. The specification is assumed correct; deviations are failures.

In practice, this comparison is difficult to achieve. Theoretically, comparison (to determine equivalence) of two arbitrary, Turing-computable functions is unsolvable. Returning to the text editor example, if the output is supposed to be "highlight a misspelled word," how can we determine that each instance of misspelling has been detected? Such difficulty is the reason why the actual-versus-expected output comparison is usually performed by a human *oracle*: a tester who visually monitors screen output and painstakingly analyzes output data. (See the "Testing Terminology" sidebar for an explanation of other common testing terms).

Two approaches to evaluating your test

In dealing with the problems of test evaluation, researchers are pursuing two approaches: formalism, and embedded test code.

Formalism chiefly involves the hard work of formalizing the way specifications are written and the way that designs and code are derived from them.⁵ Both object-oriented and structured development contain mechanisms for formally expressing specifications to simplify the task of comparing expected and actual behavior. Industry has typically shied away from formal methods; nonetheless, a good specification, even an informal one, is still extremely helpful. Without a specification, testers are likely to find only the most obvious bugs. Furthermore, the absence of a specification wastes significant time when testers report unspecified features as bugs.

There are essentially two types of embedded test code. The simplest type is test code that exposes certain internal data objects or states that make it easier for an external oracle to judge correctness. As implemented, such functionality is invisible to users. Testers can access test code results through, for example, a test API or a debugger.

A more complex type of embedded code features self-testing programs.⁶ Sometimes this involves coding multiple solutions to the problem and having one solution check the other, or writing inverse routines that undo each operation. If an operation is performed and then undone, the resulting software state should be equivalent to its preoperational state. In this situation, the oracle is not perfect; there could be a bug in both operations where each bug masks the other.

Regression testing

After testers submit successfully reproduced failures to development, developers generally create a new version of the software (in which the bug has been supposedly removed). Testing progresses through subsequent software versions until one is determined to be fit for release. The question is, how much retesting (called *regression testing*) of version n is necessary using the tests that were run against version $n - 1$?

Any specific fix can (a) fix only the problem that was reported, (b) fail to fix the problem, (c) fix the problem but break something that was previously working, or (d) fail to fix the problem *and* break some-

Software testing is often equated to finding bugs. However, test scenarios that do not reveal failures are also informative, so I offer this definition:

Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment.

The fact that the system is being *executed* distinguishes testing from code reviews, in which uncompiled source code is read and analyzed statically (usually by developers). Testing, on the other hand, requires a running executable.

A specification is a crucial artifact to support testing. It defines correct behavior so that incorrect behavior is easier to identify. Incorrect behavior is a software *failure*. Failures are caused by *faults* in the source code, which are often referred to as *defects* or *bugs*. The *oracle* compares actual output with specified output to identify failures. Generally, the code developer diagnoses the causal fault.

Software can also fail by not satisfying environmental constraints that fall outside the specification. For example, if the code takes too much memory, executes too slowly, or if the product works on one operating system but not another, these are considered failures.

Software testing is classified according to the manner in which testers perform the first two phases of the testing process. The scope of the first phase, modeling the software's environment, determines whether the tester is doing *unit*, *integration*, or *system* testing.

Unit testing tests individual software components or a collection of components. Testers define the input domain for the units in question and ignore the rest of the system. Unit testing sometimes requires the construction of throwaway driver code and stubs and is often performed in a debugger.

Integration testing tests multiple components that have each received prior and separate unit testing. In general, the focus is on the subset of the domain that represents communication between the components.

System testing tests a collection of components that constitutes a deliverable product. Usually, the entire domain must be considered to satisfy the criteria for a system test.

The second phase of testing, test selection, determines what *type* of testing is being done. There are two main types:

Functional testing requires the selection of test scenarios without regard to source code structure. Thus, test selection methods and test data adequacy criteria, described in the main text, must be based on attributes of the specification or operational environment and not on attributes of the code or data structures. Functional testing is also called *specification-based testing*, *behavioral testing*, and *black-box testing*.

Structural testing requires that inputs be based solely on the structure of the source code or its data structures. Structural testing is also called *code-based testing* and *white-box testing*.

thing else. Given these possibilities, it would seem prudent to rerun every test from version $n - 1$ on version n before testing anything new, although such a practice is generally cost-prohibitive.⁷ Moreover, new software versions often feature extensive new functionality, in addition to the bug fixes, so the regression tests would take time away from testing new code. To save resources, then, testers work closely with developers to prioritize and minimize regression tests.

Another drawback to regression testing is that these tests can (temporarily) alter the

If the code will be hard to test and verify, it should be rewritten to make it more testable.

purpose of the test data adequacy criteria selected in the earlier test selection phase. When performing regression tests, testers seek only to show the absence of a fault and to force the application to exhibit specific behavior. The outcome is that the test data adequacy criteria, which until now guided test selection, are ignored. Instead, testers must ensure that a reliable fix to the code has been made.

Related concerns

Ideally, developers will write code with testing in mind. If the code will be hard to test and verify, then it should be rewritten to make it more testable. Likewise, a testing methodology should be judged by its contribution to solving automation and oracle problems. Too many methodologies provide little guidance in either area.

Another concern for testers while running and verifying tests is the coordination of debugging activity with developers. As failures are identified by testers and diagnosed by developers, two issues arise: failure reproduction and test scenario re-execution.

Failure reproduction is not the no-brainer it might seem. The obvious answer is, of course, to simply rerun the offending test and observe the errant behavior again, although rerunning a test does not guarantee that the exact same conditions will be created. Scenario re-execution requires that we know the exact state of the operating system and any companion software—for example, client-server applications would require reproduction of the conditions surrounding both the client and the server. Additionally, we must know the state of test automation, peripheral devices, and any other background application running locally or over the network that could affect the application being tested. It is no wonder that one of the most commonly heard phrases in a testing lab is, “Well, it was behaving differently before....”

Phase 4: Measuring Testing Progress

Suppose I am a tester and one day my manager comes to me and asks, “What’s the status of your testing?” Testers are often asked this question but are not well equipped to answer it. The reason is that the state of the practice in test measurement is to count things. We count the number of inputs we’ve applied, the percentage of code we’ve

covered, and the number of times we’ve invoked the application. We count the number of times we’ve terminated the application successfully, the number of failures we found, and so on. Interpreting such counts is difficult—is finding lots of failures good news or bad? The answer could be either. A high bug count could mean that testing was thorough and very few bugs remain. Or, it could mean that the software simply has lots of bugs and, even though many have been exposed, lots of them remain.

Since counting measures yield very little insight about the progress of testing, many testers augment this data by answering questions designed to ascertain structural and functional testing completeness. For example, to check for structural completeness, testers might ask these questions:

- Have I tested for common programming errors?⁸
- Have I exercised all of the source code?¹
- Have I forced all the internal data to be initialized and used?³
- Have I found all seeded errors?¹

To check for functional completeness, testers might ask these questions:

- Have I thought through the ways in which the software can fail and selected tests that show it doesn’t?⁹
- Have I applied all the inputs?¹
- Have I completely explored the state space of the software?⁴
- Have I run all the scenarios that I expect a user to execute?¹⁰

These questions—essentially, test data adequacy criteria—are helpful to testers; however, determining when to stop testing, determining when a product is ready to release, is more complex. Testers want quantitative measures of the number of bugs left in the software and of the probability that any of these bugs will be discovered in the field. If testers can achieve such a measure, they know to stop testing. We can approach the quantitative problem structurally and functionally.

Testability

From a structural standpoint, Jeffrey Voas has proposed *testability*¹¹ as a way to determine an application’s testing complexi-

ty. The idea that the number of lines of code determines the software's testing difficulty is obsolete; the issue is much murkier. This is where testability comes into play. If a product has high testability, it is easy to test and, consequently, easier to find bugs in. We can then monitor testing and observe that because bugs are fewer, it is unlikely that many undiscovered ones exist. Low testability would require many more tests to draw the same conclusions; we would expect that bugs are harder to find. Testability is a compelling concept but in its infancy; no data on its predictive ability has yet been published.


Reliability models

How long will the software run before it fails? How expensive will the software be to maintain? It is certainly better to find this out while you still have the software in your testing lab.

From a functional standpoint, *reliability models*¹⁰—mathematical models of test scenarios and failure data that attempt to predict future failure patterns based on past data—are well established. These models thus attempt to predict how software will behave in the field based on how it behaved during testing. To accomplish this, most reliability models require the specification of an *operational profile*, a description of how users are expected to apply inputs. To compute the probability of failure, these models make some assumptions about the underlying probability distribution that governs failure occurrences. Researchers and practitioners alike have expressed skepticism that such profiles can be accurately assembled. Furthermore, the assumptions made by common reliability models have not been theoretically or experimentally verified except in specific application domains. Nevertheless, successful case studies have shown these models to be credible.

Software companies face serious challenges in testing their products, and these challenges are growing bigger as software grows more complex. The first and most important thing to be done is to recognize the complex nature of testing and take it seriously. My advice: Hire the smartest people you can find, help them get the tools and training they need to learn their craft, and listen to them when

they tell you about the quality of your software. Ignoring them might be the most expensive mistake you ever make.

Testing researchers likewise face challenges. Software companies are anxious to fund good research ideas, but the demand for more practical, less academic work is strong. The time to tie academic research to real industry products is now. We'll all come out winners. 

References

1. G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1976.
2. T.J. Ostrand and M.J. Balcer, "The Category-Partition Technique for Specifying and Generating Functional Tests," *Comm. ACM*, Vol. 31, No. 6, June 1988, pp. 676–686.
3. S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Dataflow Information," *IEEE Trans. Software Eng.*, Vol. 11, No. 4, Apr. 1985, pp. 367–375.
4. J.A. Whittaker and M.G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Trans. Software Eng.*, Vol. 20, No. 10, Oct. 1994, pp. 812–824.
5. D.K. Peters and D.L. Parnas, "Using Test Oracles Generated from Program Documentation," *IEEE Trans. Software Eng.*, Vol. 24, No. 3, Mar. 1998, pp. 161–173.
6. D. Knuth, "Literate Programming," *The Computer J.*, Vol. 27, No. 2, May 1984, pp. 97–111.
7. G. Rothermel and M.J. Harrold, "A Safe, Efficient Algorithm for Regression Test Selection," *Proc. IEEE Software Maintenance Conf.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1993, pp. 358–367.
8. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990.
9. J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Trans. Software Eng.*, Vol. 2, No. 2, June 1975, pp. 156–173.
10. J.D. Musa, "Software Reliability Engineered Testing," *Computer*, Vol. 29, No. 11, Nov. 1996, pp. 61–68.
11. J.M. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Trans. Software Eng.*, Vol. 18, No. 8, Aug. 1992, pp. 717–727.

About the Author



James A. Whittaker is an associate professor of computer science at the Florida Institute of Technology, Melbourne, and chair of the software engineering program. He is the founder and codirector of the Center for Software Engineering Research, an industry-sponsored university research lab dedicated to advancing software engineering theory and practice. His research interests are in software engineering, particularly testing and coding. He stays as far away from software process as possible. He holds a PhD in computer science from the University of Tennessee and is a member of the ACM and the IEEE Computer Society. Contact him at Florida Tech, Computer Science Dept., 150 West University Blvd., Melbourne, FL 32901; jw@cs.fit.edu.