Validating and Improving Test-Case Effectiveness

Yuri Chernak, *Valley Forge Consulting*

Effective software testing before release is crucial for product success. Based on a new metric and an associated methodology for in-process validation of test-case effectiveness, the author presents an approach to improving the software testing process.

Managers of critical software projects must focus on reducing the risk of releasing systems whose quality is unacceptable to users. Software testing helps application developers manage this risk by finding and removing software defects prior to release. Formal test methodology defines various test types, including the function test.[1] By focusing on a system's functionality and looking for as many defects as possible, this test bears most direct responsibility for ultimate system quality. Implementing the function test as a formal process lets testers cope better with the functional complexity of the software application under test. Test-plan documents and test-case specifications are important deliverables from the formal test process.[2]

For complex systems, test cases are critical for effective testing. However, the mere fact that testers use test-case specifications does not guarantee that systems are sufficiently tested. Numerous other factors also determine whether testers have performed well and whether testing was effective.

Ultimately, testers can only evaluate complete testing effectiveness when a system is in production. However, if this evaluation finds that a system was insufficiently tested or that the test cases were ineffective, it is too late to benefit the present project. To reduce such a risk, the project team can assess testing effectiveness by performing in-process evaluation of test-case effectiveness. This way, they can identify problems and correct the testing process before releasing the system.

This article describes a technique for in-process validation and improvement of test-case effectiveness. It is based on a new metric and an associated improvement framework. These work together to improve system quality before its release into production.

Evaluation = verification + validation

The evaluation process certifies that a product fits its intended use. For software project deliverables in general, and test cases in particular, evaluation commonly consists of verification and validation tasks.[3]

Verification

To start, a project team must first verify test-case specifications at the end of the test-design phase. Verifying test cases before test execution is important; it lets the team assess the conformance of test-case specifications to their respective requirements. However, such

0740-7459/00/$10.00 © 2001 IEEE

Validation serves primarily to determine whether a test suite was sufficiently effective in the test cycle.

conformance does not mean that the test cases will automatically be effective in finding defects. Other factors also determine whether test cases will be effective in the test cycle. These include design of test cases using incomplete or outdated functional specifications, poor test-design logic, and misunderstanding of test specifications by testers.

Verification activities commonly used include reviews or inspections and traceability analysis. Reviews or inspections let us evaluate test-case specifications for their correctness and completeness, compliance with conventions, templates, or standards, and so forth. Traceability matrices or trees, on the other hand, let testers trace from the functional specifications to the corresponding test-case specifications, which ensures that all functional requirements are covered by the given test cases.

Nevertheless, test cases that passed verification could have weak failure-detecting ability and, therefore, should be required to pass validation as well.

Validation

Validation can proceed as soon as testers have executed all test cases, which is at the end of the test process's test-execution phase. As its main objective, test-suite validation determines whether the test cases were sufficiently effective in finding defects.

If a test suite was effective, the system under test will likely be of high quality and users will be satisfied with the released product. But, if a test suite was not effective, there is a high risk that the system was not sufficiently tested. In such cases, the users will likely be dissatisfied with the system's quality.

If test-case effectiveness has not proved satisfactory, it is not too late to analyze the reasons and correct the test process. Using the proposed improvement framework, testers can revise and improve the test suite, and then execute the tests again. This, in turn, can help them find additional defects and thus deliver a better software product.

The test-case effectiveness metric

To perform validation objectively, testers need a metric to measure the effectiveness of test cases. When testing online mainframe systems, and especially client–server systems, a certain number of defects are always found as a side effect. By side effect, I mean the situation where testers find defects by executing some steps or conditions that are not written into a test-case specification. This can happen either accidentally or, more frequently, when the tester gets an idea on the fly.

When defining a metric for test-case effectiveness, we can assume that the more defects test cases find, the more effective they are. But, if test cases find only a small number of defects, their value is questionable. Based on this logic, I propose a simple test-case effectiveness metric, which is defined as the ratio of defects found by test cases (Ntc) to the total number of defects (Ntot) reported during the function test cycle:

$$TCE = N_{tc} / Ntot * 100\%$$

More precisely, N_{tot} is the sum of defects found by test cases and defects found as a side effect.

The proposed TCE metric might resemble Jones' defect removal efficiency metric,[4] which is defined as the ratio of defects found prior to production to the total number of reported defects. The important distinction is that DRE has the purpose of evaluating user satisfaction with the entire test process. It measures the test-process effectiveness and reflects a production or users' perspective on the test process. In contrast, my TCE metric serves specifically to validate the effectiveness of functional test cases. Unlike DRE, the TCE metric evaluates test cases from the test-cycle perspective, which provides in-process feedback to the project team on how well a test suite has worked for testers.

As I've discussed, validation serves primarily to determine whether a test suite was sufficiently effective in the test cycle. We can conclude this by comparing the actual TCE value, calculated for the given test cycle, with a baseline value. The project team selects the latter in advance, possibly obtaining it by analyzing previous successful projects that are considered appropriate as models for current and future projects. My experience with successful client-server projects delivering business applications suggests 75 perecent to be an acceptable baseline value. However, the goal for test-case effectiveness can be different for various application categories, such as commercial, military, or business applications.

When the TCE value is at the baseline level or above, we can conclude that the test cases have been sufficiently effective in a test

cycle. In this case, the project team can anticipate user satisfaction with the system in production. But, the further the TCE value falls below the baseline level, the higher is the risk of user dissatisfaction. In such cases, the project team can correct the test process based on the framework, as I'll describe.

Improving test-case effectiveness

If in-process validation finds test-case effectiveness to be less than acceptable, the project team should analyze the causes and identify areas for test process improvement.

My proposed improvement framework stems from the defect-prevention concept developed at IBM.[5] The IBM approach improves the test process on the basis of causal analysis of defects, so-called test escapes, that were missed in testing. Test escapes are "product defects that a particular test failed to find, but which were found in a later test, or by a customer [in production]."

I further evolve IBM's concept and suggest that the analysis of defects missed by test cases can help us improve test-case effectiveness. Hence, my improvement framework is based on *test-case escapes*, defined as software defects that a given suite of test cases failed to find but that were found as a side effect in the same test cycle. Once they are found by chance, we can view test-case escapes as a manifestation of deficiencies in the formal test process. Therefore, their causal analysis can help us identify areas for test process improvement.

In brief, the proposed improvement framework consists of the following steps:

1. Understand and document the test process used by the project team.
2. Make assumptions about the factors affecting test-case effectiveness.
3. Gather defect data and perform causal analysis of test-case escapes.
4. Identify the main factors.
5. Implement corrective actions.

Following these steps, either a revised part or the entire test suite (as I'll discuss later) should be executed again. As a result, testers should find additional defects that justify the improvement effort. Below I discuss each of the five steps in detail.

Clearly, my approach relies entirely on the analysis of defects missed by test cases. Consequently, it requires that a sufficient number of such defects be available. This fact can limit the applicability of the approach for some projects, for example, in the testing of mainframe batch systems. Here, testers generally exercise only preplanned conditions, and the number of defects found as a side effect is usually very low in the test cycle. But, for client–server projects that implement formal testing, the share of such defects could be from 20 to 50%, which provides a valuable source of information for test-suite validation and test-process improvement.

Let's look at the five steps.

Step 1. Understand, document the test process

When a project team uses written test-case specifications and focuses on their evaluation and improvement, this already indicates that a certain test process has been established and followed. The test process should be planned at the beginning of the software project and documented in a test plan. Commonly, testers define the test process in terms of the following phases: test planning, test design, test preparation and execution, and test evaluation and improvement.[6–8] Each phase should be planned and defined in terms of tasks and deliverables. For example, we can define the test process as follows:

- *Test planning*. In this phase, the main tasks are the definition of the scope, objectives, and approach to testing. The main deliverable is a test-plan document.
- *Test design*. This involves the design of test cases, with the main deliverables being the test-case specifications.
- *Test preparation and execution*. In this phase, preparation of the test environment, executing test cases, and finding defects are the necessary tasks, and the main deliverables are defect reports.
- *Test evaluation and improvement*. Here, the main task is analyzing the results of testing and the main deliverable is a test summary report.

In all phases, except the last, there are a number of factors that determine the effectiveness of functional test cases in a given project. Hence, the following steps of my improvement framework focus on identifying and evaluating these factors.

Step 2. Make assumptions

Once it understands and documents the test process, the project team should analyze each phase and identify factors that can affect test-case effectiveness.

Clearly, my approach relies entirely on the analysis of defects missed by test cases. Consequently, it requires that a sufficient number of such defects be available.

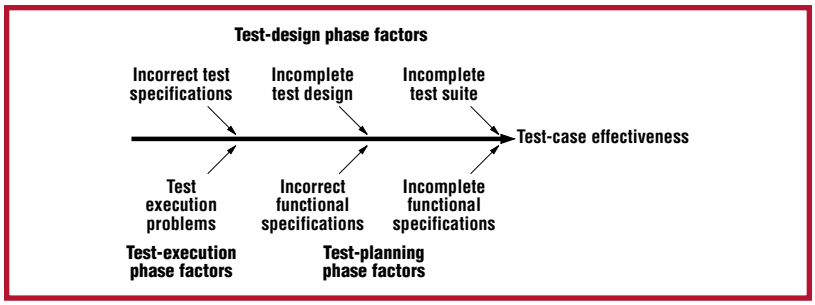

Figure 1. Factors affecting test-case effectiveness.

Test planning. The main deliverable of the test-planning phase is a test-plan document that, among other things, defines the scope and objectives of testing. We can define test objectives as features to be tested[2] that, in turn, should be traced to functional specifications. If the functional specifications do not completely define functional features, the test-plan document will not be complete either. Hence, the test cases will not completely cover a system's functionality, thereby reducing their effectiveness.

Test design. When writing a test-case specification, we usually begin by understanding and analyzing the corresponding business rule that is the object of the test. Then, we consider the test logic required for testing this functional feature. To identify necessary test cases, we can use test design techniques such as decision tables, equivalence partitioning, boundary analysis, and so forth.[1,7,8] The test-design phase can give rise to other factors that affect test-case effectiveness. First, the test suite might be incomplete and some of the business rules in the functional specifications might not be covered by test cases. Second, test-design logic could be incomplete and some of the necessary test conditions could be missing in test-case specifications. A common example of this situation is a lack of negative test cases. By definition, a test case is negative if it exercises abnormal conditions by using either invalid data input or the wrong user action. Finally, a third factor is that the test-case specifications could simply be incorrect. For example, a source document—a corresponding functional specification—could be incorrect or unclear, or there might be an error in the test-case specification itself.

All the deficiencies identified in the test-planning and test-design phases will ultimately require addition of new and revision of existing test-case specifications and their retesting.

Test preparation and execution. The test-execution phase itself can be a source of factors that reduce test-case effectiveness. For example, some test cases might not be executed or might be executed incorrectly. In addition, a tester might overlook defects, especially when the verification of expected results is not straightforward. Based on our experience, only a small number of test-case escapes stem from test-execution factors. Therefore, these factors will probably not be central to the test-case effectiveness improvement effort. However, further analysis at Step 4 might show that the proportion of defects in this category is significant. In such cases, a detailed evaluation of test-execution factors should be performed.

Figure 1 shows these factors in the form of a cause–effect diagram. I have grouped the factors according to the test-process phases in which they originate. However, at this point, they are just assumptions that should be evaluated using the following steps to identify the factors that are mostly responsible for insufficient test-case effectiveness.

Step 3. Gather defect data and perform causal analysis

To perform causal analysis of test-case escapes at the end of the test-execution phase, testers must select the defects missed by test cases. This requires the use of a defect-tracking system. Also, the testers must identify which defects were found as a result of test-case execution and which were found as a side effect—that is, as test-case escapes. Once identified and selected, the test-case escapes should be classified according to one of the factors based on the causal analysis logic shown in Figure 2.

This analysis is used to evaluate each test-case escape and understand why the test suite missed the corresponding defect during test execution. We can begin causal analysis by verifying that a functional specification has a business rule related to the given defect. If it does not, we have determined that the cause of this test-case escape is an incomplete functional specification. However, if it does, we need to check whether the test suite has a test specification that should have found this test-case escape.

If a test-case specification does not exist, this means that the test suite does not cover all business rules. Therefore, an incomplete test suite is the reason this defect was missed. If a test specification does exist, we need to check the defect against test cases in the specification. If none of them were de-

Case Study

This project was a banking application intended for external clients—financial institutions. The system had a three-tier client–server architecture with a Windows NT front-end developed in Visual Basic and Visual C++. The second tier was implemented in a Unix environment with Oracle 7 as a database engine. The third tier was a data feed from a mainframe COBOL/DB2 system. The project team consisted of 10 developers and three testers.

Because the application was intended for external clients, software quality was of great importance to project management. To ensure high quality, the project team implemented a formal test process with a focus on functional testing. The development team was responsible for functional specifications, and the test team was responsible for the test-plan document and test-case specifications. Management defined the functional testing exit criteria as follows:

- 100% of test cases are executed.
- No defects of high and medium severity remain open.
- Test-case effectiveness not less than 75%.

By the end of the test-execution phase, testers had executed all test specifications and reported 183 defects. Defects were managed using the PVCS-based defect tracking system. In reporting defects, testers classified them either as test-case escapes or as being found by conditions in test-case specifications. Testers reported 71 test-case escapes and 112 defects found by test cases. Based on these numbers, the calculated TCE metric value was 61%, which was considerably lower than the acceptable level of 75%. As a result, the project team concluded that functional testing did not pass the exit criteria and the system was likely not sufficiently tested. Hence, test-process correction and system retesting were needed.

The project team performed the test process improvement according to the framework described above. First, they analyzed all test-case escapes and classified them by appropriate causes. Next, they built a distribution of causes (see Figure A). Analysis of the distribution showed incomplete test design and incomplete functional specifications to be the main factors causing missed defects by test cases. To improve the test process, the project team began by correcting and completing the functional specifications and reviewing them with the users. A subsequent review of test-case specifications showed that the main deficiency of the test design was a lack of negative test cases. Therefore, the existing test-case specifications were completed with negative test cases.

By definition, negative test cases focus on abnormal workflow and are intended to break a system. However, the test suite initially used by the testers was not sufficiently "destructive." A significant number of defects were found as side effects as opposed to being found by conditions in test specifications. In addition, the team created a number of new test-case specifications to completely cover the business rules in the revised functional specifications. To verify test suite completeness, this time the project team used a traceability matrix, which was not done in the first test cycle. Test suite incompleteness was one of the factors that reduced test-case effectiveness (see Figure A).

After these corrections, the testers executed the revised part of the test suite. As a result, they found 48 additional defects that otherwise would have been released into production. At this point, the number of defects found during the test cycles had grown to 231. After two months in production, the rate of defects, reported by users, had noticeably declined. By the end of the second month the number of production defects was 23. The DRE metric calculated at this time was 91%, which is 231/(231+23) = 0.91, and indicated sufficient effectiveness of the test process.[4] Indeed, none of the defects reported from production by the users were of critical severity, and the users were fairly satisfied with the system quality.

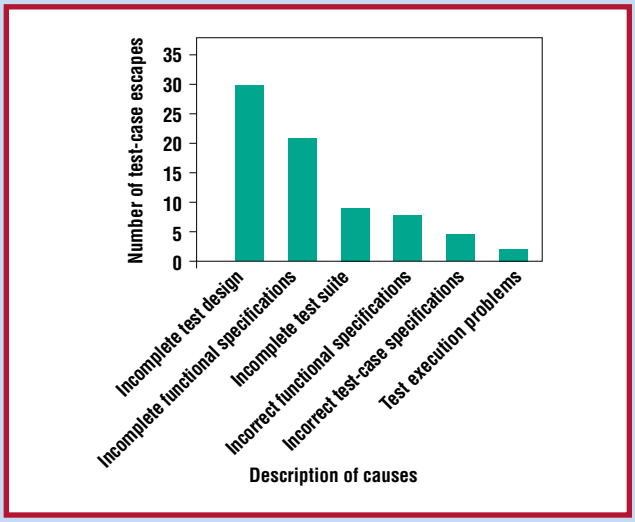

Figure A. A Pareto chart.

signed to catch such a defect, this indicates that the test specification is incomplete. Indeed, all test inputs and expected results in the test-case specification might be correct. However, the specification might include, for example, only positive test cases.

A lack of negative test cases in test specifications is a common cause of missed defects. This is a case of deficiency in test design that was used to derive test cases. Hence, we can specify that the cause of such test-case escapes is incomplete test design. But, if the test specification includes conditions related to a given defect, we need to verify that these test conditions and the corresponding expected results are correct. If they are correct, we should conclude that test-execution problems are the likely reason that the defect was missed.

If the test conditions or expected results were not correct, we need to understand why the test specification is incorrect. First, we should check the source document and see if the corresponding business rule is also incorrect. If this is the case, we should classify the cause of this test-case escape as an incorrect functional specification. Otherwise, the cause is incorrect test specification.

As a result of defect causal analysis, all

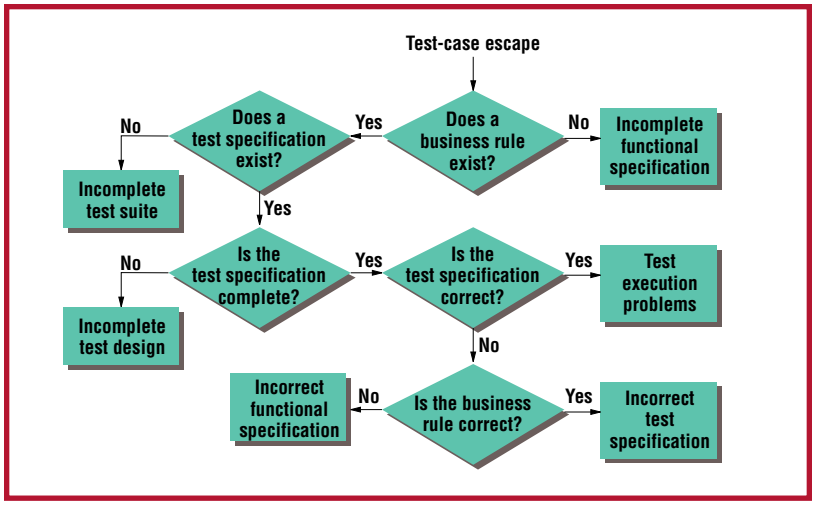

Figure 2. Test-case escape classification logic.

test-case escapes should be classified according to one of the possible causes presented in Figure 1.

Step 4. Identify the main factors

At this point, all test-case escapes have been classified according to their respective causes. We now need to identify those "vital few" factors that are responsible for the majority of the defects being missed by test cases.

For this, we can build a Pareto chart,[9] which displays frequency bars in descending order, convenient for analyzing types of problems. Once identified, the most important causes will be the focus of the next step—implementation of corrective actions.

Step 5. Implement corrective actions

After identification of the main causes of test-case escapes, the project team should implement corrective actions and repeat the test execution cycle. For the factors shown in Figure 1, corrective actions could be any of the following:

- Incomplete or incorrect functional specifications—inspect and rework functional specifications, then rework test-case specifications.
- Incomplete test suite—use a traceability matrix to ensure complete coverage of business rules by test cases.
- Incomplete test design—implement training of testers on test-design techniques; use checklists or templates to design test-case specifications; rework test-case specifications.
- Incorrect test-case specifications—inspect and rework test-case specifications.
- Test-execution problems—implement training of testers on test execution, develop and use procedures for test execution and verification of test results.

When functional specifications or test cases must be corrected, the project team should revise the test suite and execute the revised part again. However, if correction is required only due to the test-execution problems, the same test can be used for retesting. The main objective of retesting is to find additional defects. If additional ones are found, this fact can justify the whole improvement effort.

The "Case Study" box illustrates how my proposed approach to test-case effectiveness validation and in-process improvement was implemented in a client–server project.

This technique for in-process validation of test cases is intended to give project teams better visibility into test-process effectiveness before their systems are released into production. The proposed technique can be applied within any project management model, including incremental or evolutionary models, where it can be used for assessment of test-process effectiveness and its tuning from one incremental build to another.

A project team has to decide in advance what level of test-case effectiveness is acceptable for their project. Such a requirement can vary depending primarily on the project's criticality. Future work will focus on developing a formal approach to selecting a baseline value for the TCE metric.

About the Author

Yuri Chernak is president and founder of Valley Forge Consulting, Inc., a consulting firm that specializes in the field of software quality assurance and systems testing. He has over 20 years of experience in the software industry. As a consultant, he has worked for various clients, primarily for the brokerage firms in New York. He has a PhD in computer science and is a member of the IEEE. His research interests cover systems test methodology, software metrics, and process improvement. He has been a speaker at international conferences on software quality. Contact him at ychernak@idt.net.

Acknowledgments

I am grateful to Vladimir Ivanov for his help in preparing this material. I thank Richard Reithner for editing the article. Finally, I am grateful to the *IEEE Software* reviewers for their helpful feedback and comments.

References

1. G. Myers, *The Art of Software Testing*, John Wiley & Sons, Upper Saddle River, N.J., 1979.
2. *IEEE Std. 829-1983, Software Test Documentation*, IEEE, Piscataway, N.J., 1983.
3. *IEEE Std. 1012-1986, IEEE Standard for Software Verification and Validation Plans*, IEEE, Piscataway, N.J., 1986.
4. C. Jones, *Applied Software Measurement*, McGraw-Hill, New York, 1991.
5. R. Mays et al., "Experiences with Defect Prevention," *IBM Systems J.*, vol. 29, no. 1, 1990, pp. 4–32.
6. Y. Chernak, "Approach to the Function Test Decomposition and Management," *Proc. 15 Pacific Northwest Software Quality Conf.*, PNSQC/Pacific Agenda, Portland, 1997, pp. 400–418.
7. E. Kit Longman, *Software Testing in the Real World*, Addison-Wesley, Reading, Mass., 1995.
8. P. Goglia, *Testing Client–server Applications*, QED Publishing Group, Wellesley, Mass., 1993.
9. L.J. Arthur, *Improving Software Quality*, John Wiley & Sons, Upper Saddle River, N.J., 1993.