Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Test Management

Panos Louridas

In many projects, testing consumes the single biggest amount of resources of all activities. We tend to collect test cases like stamps without clear strategy—just in case. Many companies suffer with insufficient quality, visibility, and test progress management. Author Panos Louridas introduces test management. I look forward to hearing from both readers and prospective authors about this column and the technologies you want to know more about. —*Christof Ebert*

To vouch this, is no proof,
Without more wider and more overt test
Than these thin habits and poor
likelihoods
Of modern seeming do prefer against him.
—*Othello*, Act I, Scene III

MICROSOFT WROTE MORE than 1 million test cases for Office 2007. This figure might seem extreme or only appropriate for leviathan-sized projects, but the fact is that test cases constitute a component of software that developers can ignore only at their peril. Test management deals with how we

- navigate through a proliferation of test cases,
- organize them,
- assign them to testers,
- orchestrate testing,
- collect test results, and
- measure progress.

Several interesting technologies have recently been introduced for managing test cases, which I'll describe in this column. To understand the stakes involved, it's worth seeing how much effort testing requires.

Testing Effort

Test management makes sense, given the high cost of testing. The value of managing test cases derives directly from the amount of effort that testing takes up in the project. Industry figures from Steve McConnell and Christof Ebert show the following:[1,2]

- System testing requires from 16 to 29 percent of the total effort, depending on the project's size (see Table 1).
- Test effort varies depending on the nature of the project (for example, safety-critical projects need more than a simple gadget), complexity, and, of course, processes and tools (see Table 2). For instance, with insufficient input quality, test effort immediately increases by a factor of two or three.
- Systematic test planning, frequent test coverage reviews, incremental development, daily builds, and smoke tests with automatic regression are among the biggest efficiency levers in industry.

For such effort levels, it behooves project managers and developers to make testing as efficient as possible. This requires measuring tests, to which we turn next.

0740-7459/11/$26.00 © 2011 IEEE

TABLE 1

Total effort breakdown for projects of different sizes for application software.

Size (in KLOC)	Activity				
	Requirements	Architecture and planning	Construction	System test	Management, overheads
1	4%	10%	61%	16%	9%
25	4%	14%	49%	23%	10%
125	7%	15%	44%	23%	11%
500	8%	15%	35%	29%	13%

TABLE 2

Examples of developer-to-tester ratios.

Environment	Observed developer-to-tester ratios
Common business systems (internal intranet, management information systems, and so on)	3:1 to 20:1 (often no test specialists at all)
Common commercial systems (public Internet, shrink wrap, and so on)	1:1 to 5:1
Scientific and engineering projects	5:1 to 20:1 (often no test specialists at all)
Common systems projects	1:1 to 5:1
Safety-critical systems	5:1 to 1:2
Microsoft Windows 2000	1:2
NASA space shuttle flight control software	1:10

Test Measurement

Test cases, in addition to helping increase a product's quality, are valuable in improving the development process when used to measure a series of process metrics that incorporate test results.

Defect removal efficiency (DRE), or test effectiveness, is the ratio of the number of defects found during testing to the total number of defects found. For instance, if 90 bugs are found by tests and 10 bugs are discovered by other means (later in the product life cycle, by users, and so on), the test effectiveness is 90 / (10 + 90) = 0.90.

Test efficiency is a metric that the literature has confused with test effectiveness—perhaps the semantic proximity of efficient and effective are to blame. *Fowler's Modern English Usage* defines *effective* as "having a definite or desired effect," and *efficient* as "productive with minimum waste or effort." Efficiency, therefore, is a measure like power in physics. Even with this clarification, *test effectiveness* encompasses different definitions in the literature. In the broadest sense, it's the number of defects found divided by the effort expended to find them. This can be counted, for instance, by the number of defects found divided by the number of test cases. A different definition is the percentage of tests that find bugs—it can also show which tests are more effective in finding bugs.

Pass rate is the ratio of tests that pass to the total number of tests.

Passes versus failures is the ratio of tests that pass to the number of tests that fail. This can be useful to show failures in projects with a large number of tests. For instance, for 1 million tests, a 95 percent pass rate is equivalent to 50,000 failed tests—that is, a passes versus failures rate of 19.

A *test progress S curve* is a graph plotting time units—for instance, weeks—on the x-axis and the number of tests to be completed successfully, attempted, and passed at each time unit on the y-axis.

The graph is called an S curve because it follows this shape: the test numbers that are plotted are cumulative, and the number of planned tests starts low, then increases, and finally levels off as it gets closer to release time (see Figure 1). The test progress graph is used to guide the development effort, highlighting delays in testing effort (when the number of tests attempted is less than the planned number) or in project quality (when the number of passed tests lags significantly behind the planned number).

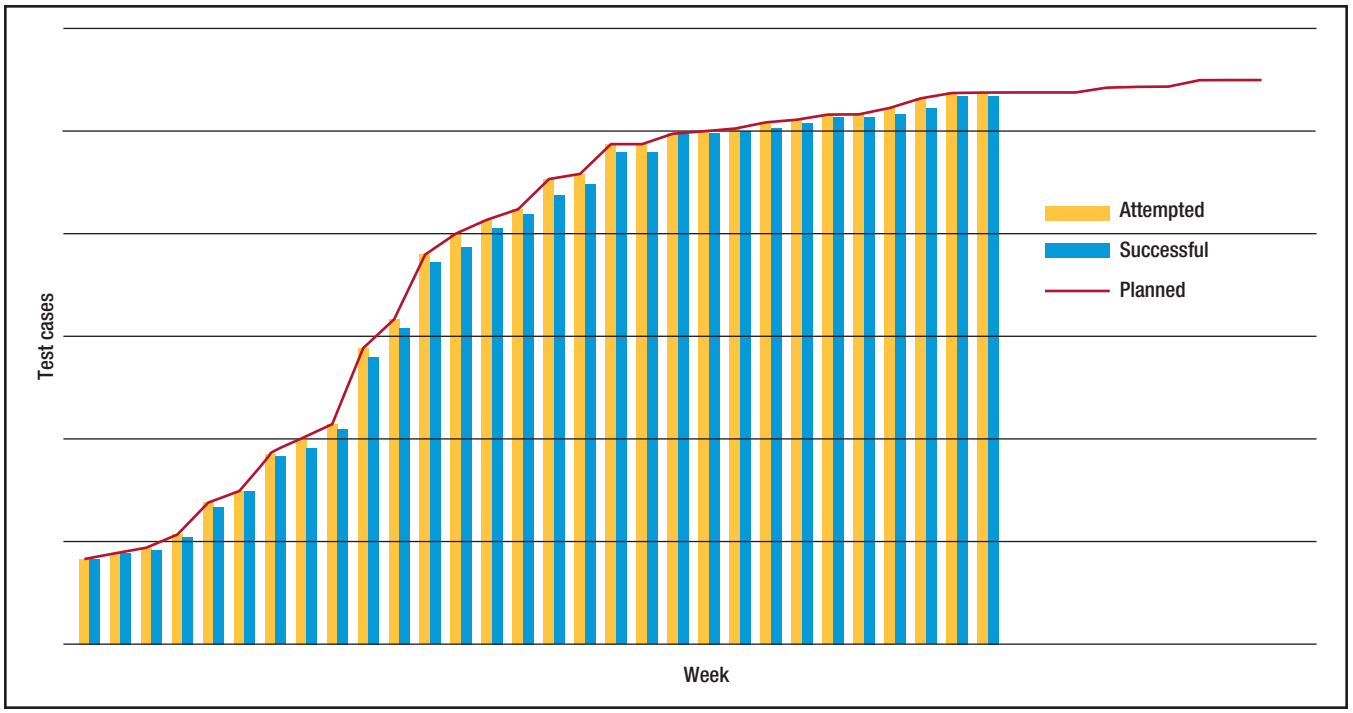

FIGURE 1. Sample test process S curve.[3] Cumulative graph of attempted, successful, and planned test cases over time.

Testing defect arrivals over time is also a graph, plotting time units on the *x*-axis and defect arrivals, discovered by testing, on the *y*-axis. The graph should normally be a hump-like shape: in the project's beginning, few defects are reported, as there are few tests and few opportunities for defects anyway; as the project progresses, both tests and discovered defects pile up; but as we get closer to shipping, fewer defects should show up.

Testing defect backlog over time is again a graph, plotting the time unit before the shipping date on the *x*-axis and defects discovered by testing and not yet resolved, therefore remaining in the backlog, on the *y*-axis. The metric can be used to guide process planning by agreeing in specific targets for defect backlog before shipping.

Test coverage measures how extensively tests cover the code to be tested. It can be measured in different ways—for example, with the ratio of the number of test cases executed divided by the number of test cases required to test all functions in the code, or by the number of branches tested divided by the total number of branches in the program. Test coverage is useful in itself to check that there are no blind spots left in the program, but it's also useful in deriving other metrics that relate to project process progress.

Test confidence is a process metric that uses test coverage. It is defined by the following formula:

$$TestConfidence = \left\{1 - \frac{ErrorsFoundInLastTest}{TestCasesRun}\right\} \times TestCoverage.$$

So, if we ran 400 test cases in total and discovered three errors in the last run, while the test coverage was calculated to be 95 percent, the test confidence would be

$$\left\{1 - \frac{3}{400}\right\} \times 0.95 = 0.94.$$

Test coverage has been used to define test efficiency in terms of tester effort:

$$TestEfficiency = \left\{1 - \frac{TesterDays}{Errors + TestCases}\right\} \times TestCoverage.$$

Continuing with the previous example, if we had 400 test cases in total, executed in 180 person days, and the test cases found 260 errors, while the final test coverage was 0.95, the test efficiency would be

$$1 - \left\{\frac{180}{260 + 400}\right\} \times 0.95 = 0.69.$$

Test Case Management Tools

For a test case to be useful, it should include

- the goal, describing briefly what it's purported to test;

TABLE 3

Test case management tools.

	Seapine TestTrack	QMetry	TestRail	XStudio
Interoperability with issue-tracking systems	Yes (with Seapine TestTrack Pro)	Yes, with Mantis, Bugzilla, and JIRA Enterprise Edition	Yes, through URLs	Uses its own bug-tracking database, connectors with JIRA, Trac, Bugzilla, and Manti
Interoperability with authentication/ authorization systems	Yes, it can use Single Sign-on	No	Yes, it can use Single Sign-on	No
Integration with requirements	Yes, via TestTrack RM	Handles requirements internally	Yes, through URLs	Handles requirements internally
Integration with source control tools	Seapine Surround SCM, CVS, Clearcase, PVCS, Perforce, SourceOffSiteClassic, StarTeam, Subversion, and Visual SourceSafe	No	No	No
Integration with developer tools	VisualStudio and Eclipse	No	No	No
Interface	Client program, Web interface (for subset of functionality), and SOAP SDK	Web interface	Web interface and API for submitting test changes and test results	Client program, Web interface, and SDK for integrating with existing tests
Platforms	MS Windows, Mac OS, and Linux	Available as hosted software as a service; on-premise installation MS Windows and Linux with PHP and MySQL (for instance, XAMPP)	MS Windows server 2003 or 2008, IIS with FastCGI/PHP integration; Unix-based OS with Apache, MySQL, and PHP	Java Runtime Environment 1.6 for the server, MS Windows, Linux, Mac OS fat clients, and Web client requiring Web server (Apache, Tomcat, IIS)
Databases	Internal, MS SQL, Oracle, PostgreSQL, and MySQL	MySQL	MS-SQL and MySQL	MySQL
License	Proprietary	Proprietary	Proprietary	Free; parts are open source

- the rationale, explaining why it's important;
- the preconditions, listing the constraints on the environment that must be in place for it to run;
- the inputs;
- the steps, in numbered sequence unless it's implemented in code;
- the expected results;
- instructions on how often and when it should run; and
- configurations under which it should run.

The simplest way to manage test cases is with a spreadsheet containing the following information:

- The name of the test suite, and for each test suite, the name of the test.
- The state of each test case, with values pass, fail, or no value if the test is queued for execution.
- An identifier for the system configuration used for the test case. The identifier is used for looking up the configuration details in a separate spreadsheet.
- The identifier of the bug discovered by the test. If the test discovered more than one bug, they can be listed with commas in a single cell or in different rows (with the other

FURTHER READING

A good overall discussion on testing in big settings is *How We Test Software at Microsoft* (A. Page, K. Johnston, and B.J. Rollinson, Microsoft Press, 2008).

The figures and tables in this column on testing effort are from *Software Estimation: Demystifying the Black Art* (S. McConnell, Microsoft Press, 2006).

A good discussion on test metrics and how to use them to guide the software process is *Metrics and Models in Software Quality Engineering* (S.H. Kan, 2nd ed., Addison-Wesley, 2003). Much of the material for test metrics used in this column comes from this book.

Although targeting agile practices, *Agile Testing: A Practical Guide for Testers and Agile Teams* (L. Crispin and J. Gregory, Pearson, 2009) is of wider relevance and offers valuable practical advice. The book presents an alternative categorization of tests in quadrants than the one used in this column.

Chapters 8 and 9 of *Software Engineering Theory and Practice* (S.L. Pfleeger and J.M. Atlee, 4th ed., Prentice Hall, 2009) provide an introduction to testing and test management.

"Validating and Improving Test-Case Effectiveness" (Y. Chernak, *IEEE Software*, vol. 18, no. 1, 2001, pp. 81–86) presents ways to improve testing efficiency (although the author prefers the term effectiveness).

A detailed discussion on metrics, including test metrics, is *Software Measurement: Establish—Extract—Evaluate—Execute* (C. Ebert and R. Dumke, Springer, 2007), from which I drew the material in this column, the formulas for test coverage, test efficiency, and test confidence, and data for test costs.

IEEE Standard 829-2008: Software and System Test Documentation (a revision of the earlier *IEEE Standard 829-1998*) treats the use and contents of test documentation, test tasks, required inputs and outputs, master test plans, and level test plans, and should be of interest to companies and organizations that are serious about testing.

For more details on how to use a spreadsheet for managing test cases, as well as other practical information, see *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing* (R. Black, Wiley, 2002).

Effective Software Testing: 50 Specific Ways to Improve Your Testing (E. Dustin, Addison-Wesley, 2002) presents specific items of advice on testing.

A very good presentation of what makes a good test case (which influenced this column) is "What Is a Good Test Case?" (C. Kaner, 2003, available at http://www.kaner.com/pdfs/GoodTest.pdf).

fields copied from the row above).
- The tester's initials.

Although this might sound simplistic, you might take into account how many tests are conducted without the use of a spreadsheet. It's easy to think of ways to improve the spreadsheet and make it part of the development process. There's no reason why it should remain a personal spreadsheet—put it up on a team environment like Google Docs or a team wiki.

A host of more sophisticated approaches to managing tests exists, as well as a wealth of tools, both open and closed source commercial ones. (Tellingly, at one point there were at least a half-dozen test case management systems in MS Windows at Microsoft, but then teams started migrating to the test case management tools in the Visual Studio Team System). Things to consider when choosing which to adopt include the following:

- *Interoperability with bug- or issue-tracking systems.*
- *Interoperability with requirement-tracking systems.* Ideally, a test case management tool should offer a three-way linkage between functional requirements, test cases, and bugs. In this way, we can know which test cases correspond to a requirement, identify requirements that don't have test cases, or map bugs to specific requirements.
- *Information kept for each test case.* This information should include the material we listed in the beginning of this section and can also include information such as the date the test was executed, the test author, links to related test cases, and trace logs.
- *Degree of automation.* You could simply use the tool for housekeeping test runs and results (like the spreadsheet solution or a simple wiki-based one) or guide the entire test process by invoking test plans.
- *Reporting capabilities and metrics calculation.* The metrics can include the ones presented earlier, and the results should be able to be presented in reports. Even better, the metrics results and the underlying data can be available through an API.
- *Interoperability with authentication systems.* Ideally, each project participant should have a single identity and should use that identity for everything, be it committing code, announcing bugs, running tests, or even writing in project-related mailing lists.
- *Installation requirements.* Usu-

ally these tools are deployed on a server so that they're available to the development team over the Web. They might be built on popular Web development frameworks (like Ruby on Rails), but in general it's worth checking how well they integrate with your own ecosystem of tools and infrastructure.
- *Cost and license.*

Test case management tools typically start with a way to define test cases, for instance, with a series of templates where the user fills in the fields that constitute the test case (see Figure 2). It might also prove possible to establish relations between test cases, to place conditions on their execution (for example, test case A should be executed if test case B succeeds), and to automatically generate variants of test cases depending on variables (such as system configuration). Test cases are then assigned to users. Overall progress and an execution overview is provided through some means of a dashboard that presents the test results both in graphics and numbers. Based on these numbers, we can compute test metrics; tools also prepare reports that can be useful for management purposes.

The most important attribute of a tool isn't its technical capability but how effectively it's used and how well it's adopted and used throughout the development process (see the sidebar "Further Reading"). Promoting a culture of defect prevention and quality assurance is more important than any specific tool that Othello would deem "modern seeming."

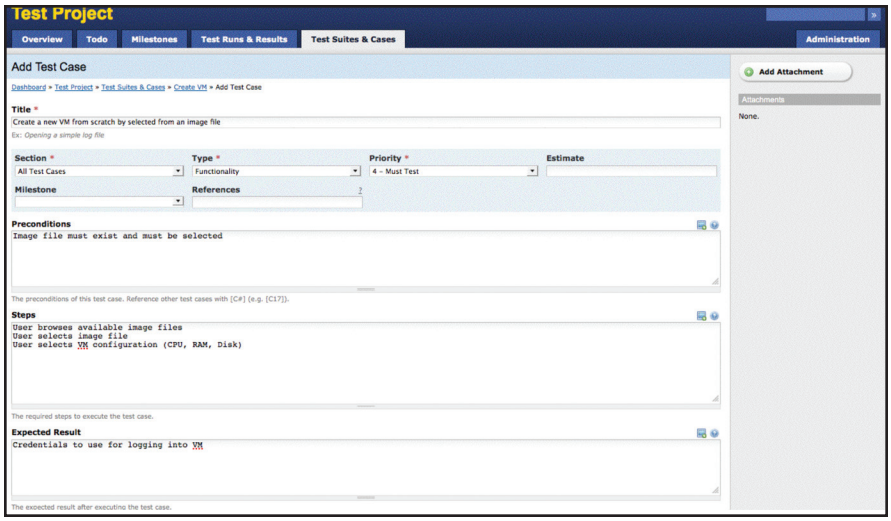

FIGURE 2. Adding a test case in TestRail. The user fills in test case information like type, priority, preconditions, and steps.

References

1. S. McConnell, *Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006.
2. C. Ebert and R. Dumke, *Software Measurement: Establish – Extract – Evaluate – Execute*, Springer, 2007.
3. S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2003.

PANOS LOURIDAS is a consultant with the Greek Research and Technology Network and a researcher at the Athens University of Economics and Business. Contact him at louridas@grnet.gr or louridas@aueb.gr.