CUTTER IT JOURNAL
The Journal of Information Technology Management

CUTTER CONSORTIUM
ACCESS TO THE EXPERTS

Reprint

Right Up Front: Strategies for Prioritizing Test Activities

by Jim Brosseau

opinion

Right Up Front: Strategies for Prioritizing Test Activities

by Jim Brosseau

Software testing is often one of the most challenging issues IT management has to contend with. While many organizations have a quality assurance group, in practice these are usually quality control teams whose primary activity is to exercise the end product against whatever forms of product requirements are available. This practice is both cause and effect for many of the challenges that management faces.[1]

People that are dedicated to these test activities often find themselves with relatively little to do until late in the project lifecycle, when the product is coming together and can be integrated and tested as a whole. The resource leveling issues this creates are often resolved by shuffling resources from project to project as they are required, or by outsourcing end-product testing activities to external organizations. Both of these practices reduce the effectiveness of the test team, as its members will have less intimate knowledge of the product and the development team. Such practices also increase the dependence on high-quality documentation as a basis for testing.

With little or no insight into the product deliverables until late in the lifecycle (when sufficient code has been generated for testers to have at it), there is a strong likelihood that the project will lapse into the "90% done" syndrome. That is, the project will quickly progress, apparently on schedule, until the product undergoes its first real assessment. At that point, it is often found that those activities that were deemed to be complete now need to be revisited, quickly consuming any contingency that was built into the schedule.

After contingency, the time allocated to test activities, which are often very poorly defined, is the next to suffer. Project scheduling for test activities frequently consists of inserting a placeholder task after code complete for the entire breadth of test activities. This is a high-risk approach whether or not the schedule is reasonably managed throughout the project, for as the deadline approaches, it is easy to rationalize the cutting of this poorly defined chunk of time in order to get the product out the door on time.

Prioritization of testing can be considered at several levels. Most discussion in the literature concerns appropriate selection of test cases within a given test phase, such as using test equivalence to select specific cases when unit testing a unit of code. This article expands the interpretation of "test" to describe the verification of any artifact that is created in the overall project lifecycle, and thus prioritization refers to the selection of appropriate activities throughout the lifecycle that will increase the overall likelihood of project success. This expanded interpretation is necessary to take advantage of the prioritization schemes discussed below.

Clearly, there are compelling reasons to adjust our approach to software product testing. The good

With little or no insight into the product deliverables until late in the lifecycle, there is a strong likelihood that the project will lapse into the "90% done" syndrome.

[1]Please note that this article does not cover allocation of specific activities to individuals or groups, as there is no generally accepted alignment for this across the industry. Rather, it provides a broader taxonomy of activities that could fall under the category of testing, regardless of who in the organization is assigned these activities.

©2002 Cutter Information LLC

news is that doing so is highly likely to result in not more time and money spent, but a reduction of both, along with reduced risk, easier scheduling, and a higher-quality product.

EXPANDING THE DEFINITION OF TEST ACTIVITIES

The V-model is commonly used as a vehicle for communicating the different types of testing that can be performed on a software product. Down the left-hand arm of the V, we have the traditional technical phases of development, usually captured as requirements, design, and coding. Corresponding with each of these on the way up the right arm of the V are the three major types of test activities that can take place: acceptance testing, integration testing, and unit testing. (These are shown in Figure 1 as "(a)" activities.)

The model is sometimes presented with additional layers of business requirements or architectural design and corresponding verification activities, but it is substantially the same.

One of the problems with the V-model is the implied waterfall lifecycle within its structure. However, the relative merits of different lifecycles need not concern us here. I would simply note that the models I describe do not imply any specific lifecycle, other than to suggest that it is best to understand what you are doing and how you will do it *before* you do it, and then confirm you did it. This principle applies to the classic waterfall as well as to any of the iterative derivatives of the waterfall. Remember that any model is presented to communicate a specific point and will have inherent limitations. As the statistician George Box noted, "All models are wrong; some models are useful."

There is an implied temporal relationship from left to right across the V-model in Figure 1, as well as a relationship between the development phases and types of test activities. Unit testing is done against small portions of code, integration testing is done to confirm that the design and architectural decisions are sound, and acceptance testing is done against the specification to

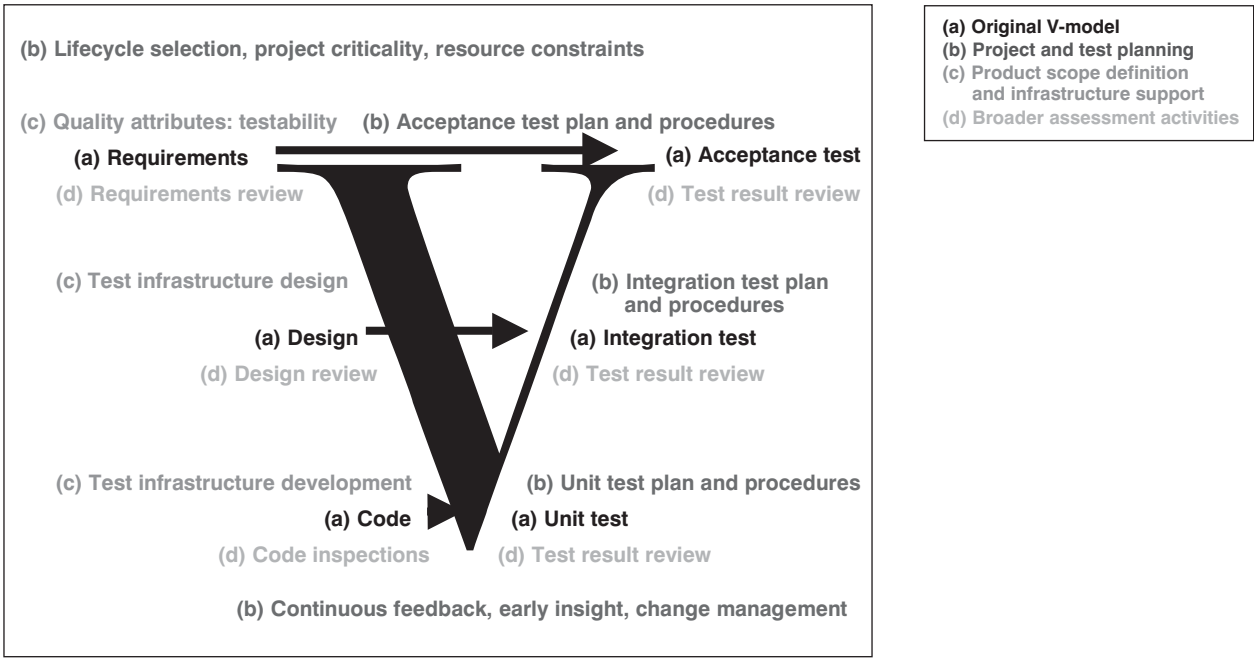

Figure 1 — The V-model for testing (a), expanded (b through d).

confirm that the product does what was intended.

While this enhanced model provides a clear perspective of these activities, we have seen that the initial V-model itself often entails far more testing than software product companies (particularly the small entrepreneurial shops) are willing to consider in product development. Especially as pressures increase, development shops fall into the "code-and-fix" mode of operation, resulting in a late, poor-quality product. Whole product testing via the user interface ("GUI bashing"), which is often the only test approach taken, makes it virtually impossible to adequately exercise the application in its entirety. At face value, the V-model perpetuates the notion that testing is limited to the dynamic testing of code-related artifacts created in a project.

Time spent in effective planning can drastically reduce the inefficient effort and rework downstream on a software project.

There are a large number of additional activities that can be added to an "augmented V-model" for test, which is the first step in building up a more mature test strategy for your product. These fall into the broad categories of:

- Project and test planning
- Product scope definition and infrastructure support
- Broader artifact assessment activities

Project and Test Planning

Time spent in effective planning can drastically reduce the inefficient effort and rework downstream on a software project, and one of the key areas in planning is the allocation of time for test activities.

Product criticality — whether the product being built is a knockoff prototype (such as a proof-of-concept for a new technology or architectural concern) or a mission-critical system (such as an air traffic control system) — is the primary driver for selection of test activities throughout the lifecycle. It is important to consider external product criticality for the end user as well as for the business. How important is this product to the company's business plan and overall success?

Another key activity is the selection of project lifecycle, based on factors such as scope volatility, technical uncertainty, schedule aggressiveness, and others. Determining how the product will be developed — using a traditional waterfall approach, one of the emerging agile approaches, or something in between — will be crucial in deciding which test activities are appropriate and how often they will be performed through the overall product cycle.

The third important area to consider is the project constraints and priorities. The project triangle, usually described as tasks, time, and resources, is really incomplete. It should be more accurately defined as tasks, resources (including time), and quality. What generally happens when projects get squeezed is that quality, which as noted is rarely a defined entity for software projects, is implicitly sacrificed in order to meet the overwhelming demands on tasks and resources. Corners are cut and the project is delivered, but what is the overall cost? Typically, one of these dimensions is absolutely critical for project success, and the others are more malleable.

The value of generating specific test plans and procedures as early as possible in the product lifecycle is twofold. First, these artifacts can be used as a strong validation of the information from which they are generated. For example, integration test plans can expose testability deficiencies in how the product specification was written. Second, this earlier-phase activity can be an effective utilization of test resources [2, 3].

Product Scope Definition and Infrastructure Support

A key component of defining product scope that is often overlooked or done poorly is the definition of product quality attributes, or non-functional requirements. (These

©2002 Cutter Information LLC

are also known as the "ilities," as most taxonomies of these attributes end in that suffix — testability, usability, and so on.) Capturing and maintaining a clear understanding of the known scope (functional and quality) for a project is critical for success. While early views of scope may be fraught with uncertainty, particularly in evolutionary projects using agile development approaches, ensuring that this scope is commonly understood is essential.

As both the internal business and end-user criticality of the software being developed increases, there will likewise be an increasing need to be able to:

- Repeat tests over time to form a regression suite that will confirm that changes do not adversely affect other components of the system
- Have consistency in test results to simplify interpretation of results across the development and test teams
- Automate the tests to reduce the time required to perform tests, especially as they are run repeatedly over time
- Reduce the nonproductive test time or that time required for testers to get the system to the point where they can exercise the functionality they are interested in
- Reuse infrastructure elements from project to project for increased savings for the organization

Often, as a result of determining that testability is a high-priority attribute for the system, a specification of the required test infrastructure will be generated. This infrastructure can be as simple as a few hooks in the system to provide deeper insight into the internal workings of the system with internal auditing and logging capabilities, or it can be a complex, comprehensive system of frameworks, test scripts, scaffolding and stubs, and external tools to improve overall testability. There are many resources available today that will allow you to expedite the construction of an effective infrastructure, from commercial testing tools to the X-unit family of open-source test frameworks.

Early project decisions invariably have a greater impact than those made at a later date.

Broader Assessment Activities

As noted above, many projects are challenged because the first real look into progress takes place when the software comes together and the product can be assessed as a whole. It is important to note, however, that anything that is generated as part of the project can be subjected to test: product visions, business cases, product scope, plans, schedules, designs, test results. The earlier validation is done the better, as there is a fan-out of impact. Early project decisions invariably have a greater impact than those made at a later date.

It has been variously shown that the cost of removing a defect from a product increases dramatically as a function of how long the defect resides in the system [1, 4]. The numbers are compelling, and there are several factors that contribute to that increasing cost:

- As the defect advances beyond the phase in which it was injected, the product often involves a larger group of people, and the corresponding effort and communication required to clean up the problem increases accordingly, as defect repair becomes a multistep process.
- The longer the defect is in the system, even if it remains within its current phase, the greater the likelihood that additional product will be built on top of the defect. This is the primary reason why there is a high probability that new defects will be injected as a result of repairing existing defects. While Microsoft was generally criticized in the press for the high number of known defects that shipped with Windows 98, it's actually a sign of Microsoft's maturity that it had found the defects and acknowledged the risk associated with fixing these bugs.
- As defects are discovered late in the cycle, it is often

the case that the required fix time can delay product deployment, potentially resulting in lost opportunity (although this would have to be balanced against the cost of a defect being discovered by an end user).

- If a significant defect remains in the product to be discovered by the end user, the costs for repair increase dramatically, as we include the costs associated with the additional release cycle as well as the corresponding damage to customer confidence.
- For mission-critical systems, these last two factors may entail additional costs such as penalties or loss of life.

By expanding our view of test activities to include such things as software inspections and reviews of documents, we can balance test resource loading throughout the lifecycle.

Several sources report that software inspections are a high-leverage approach that can be used to improve product quality and reduce risks [1, 5]. By expanding our view of test activities to include such things as software inspections and reviews of documents and test results, we can balance test resource loading throughout the lifecycle. Artifacts can be verified sooner for earlier insight into true project progress, and these static testing approaches will capture a different class of defects than can be captured through dynamic testing.

These static tests of early project artifacts can focus on a number of attributes to ensure that defects are discovered and a stable basis for ongoing development is created. A software specification, for example, is the central source of information for all stakeholders in a project. It should be complete, consistent, understood, agreed upon, and followed by everyone in the organization. It should cover both the functional capabilities of the product as well as the nonfunctional capabilities, both captured in a testable fashion. It should be the basis of estimates, development activities, and marketing projections. It needs to be managed, not only in its original creation, but in the changes to the specification throughout the life of the project. There is likely a need for the contents to be prioritized, and all stakeholders must agree on the content: development, quality assurance, marketing, project management, and the end-user representative. All of these attributes can be assessed, and many apply to other early project artifacts that are generated.

TEST PLANNING STRATEGIES

Balance is the watchword when identifying how to allocate test resources over the project lifecycle. Throwing testers at the product near the end of a project is rarely an adequate solution. In the expansion of the V-model described earlier, it must be made clear that it is highly unlikely that all of these additional activities could (or should) be performed in their entirety. The expanded model in no way implies that additional time or resources will be allocated to the test effort; it simply expands the breadth of activities that you need to consider when allocating your precious available time.

Despite the pressure found on many projects to show early "product" in the form of source code and executables, it is generally acknowledged that a reasonable amount of up-front planning will more than pay for itself over the project lifecycle through the reduction of risk, efficient utilization of resources, and alignment of the project team. Planning of test strategies and activities is a key component of this time.

There are several items that need to come from the initial project planning stage to effectively drive test effort prioritization. Mission criticality, both for the end user and for the company, will determine the level of effort required, as well as the priority of product quality in the triangle of tasks, resources, and quality.

Most of us are familiar with the curves showing the escalation of cost as defects remain in the system. Both this and the 90% done syndrome are compelling arguments that the best insight into product quality is insight gained

©2002 Cutter Information LLC

as early as possible in the project lifecycle. While many projects focus only on code assessment, and often only as the complete product is integrated, bringing the assessment activities up front is very valuable.

When planning test activities, it is most appropriate to prioritize activities in terms of their potential to mitigate risk on the project. In this context, risk is defined as a combination of the impact of an event on a project (here, in essence, the damage caused by a defect) and the likelihood that the defect will occur.

Prioritize activities in terms of their potential to mitigate risk on the project.

In addition, a breadth-first approach is critical when distributing test resources to the potential activities. Rather than attempting to thoroughly exercise one aspect of testing (such as coverage in unit testing or code inspections of all software), consider an audit-like approach to all aspects of validation activities as the basis for moving forward. You will likely find that if these audits are appropriately selected (based on the areas of prioritization listed below), several aspects of validation will expose deeper problems and warrant additional investigation. This exercise will help you determine how best to allocate the "placeholder bin" of test time in the original schedule.

PRIORITIZATION CRITERIA

Of all the entities captured on the expanded V-model above, the up-front activities of planning are the most crucial — and the most dangerous to ignore. It is inadequate planning that usually leads to the common problem of testing being only an end-product GUI-based assessment of quality, in a very compressed time frame, with the expected poor results.

Determining the testability requirements for the product is key to proper prioritization of subsequent test activities. If the product is sufficiently critical to warrant a comprehensive test strategy, a testability needs assessment can generate reasonable clarity for determining downstream activities. The following criteria can be used to determine the relative priority of these downstream activities.

Complex Algorithms and Logic

These areas of the code are extremely difficult to exercise completely from the external user interface, as it is usually necessary to use a variety of boundary conditions and valid and invalid data to ensure the integrity of the results. *Unit tests* that focus strictly on the algorithm (or a portion thereof) with supporting infrastructure to simplify data input and results analysis are often vital for thorough testing. If the algorithms are based on published formulas, *deskchecks*

A breadth-first approach is critical when distributing test resources to the potential activities.

(inspections) can be used to test the implementation, preferably against several sources if available. There are often several ways to implement the same algorithm, and these can be developed separately in a *prototyping* exercise to ensure the validity of the results.

Core Functionality

Elements of the system that are (a) essential components of overall system capabilities, (b) central elements of the architecture that serve as funneling points for data or processing, or (c) base components upon which the remainder of the system is built are important points of focus. Paying attention to them will help minimize the risk of critical failures or of building on top of flawed components. Focusing on these areas with a *reusable unit test infrastructure* will ensure that this core remains correct throughout the evolution of the product, and an *initial iteration* that focuses on this core functionality can be used to create a valid basis before expanding functionality.

Critical Relationships with External Systems

The interfaces with external systems are often more critical than the user interface, as these external systems are typically less flexible, and errors in form or function will

often result in system failure. These interfaces may warrant *specialized test harnesses* in order to verify that both the external system and the product being developed are working as specified, in an environment where they can be tested independently of one another. In addition, *document review* and *change management of the interface specifications* are important for capturing issues before they are implemented.

Inspections add value by providing a vehicle for reinforcing team best practices, disseminating product functionality, and increasing team cohesiveness.

Human Resource Issues

Inexperienced developers and testers or people new to the team can be supported through *increased code inspections* or *mentoring* to reduce the additional risks associated with their inexperience and/or lack of familiarity with the system. The value of code inspections, if performed properly, goes well beyond the early discovery of defects. For both the reviewer and the author of the product being reviewed, inspections add value by providing a vehicle for *reinforcing team best practices*, *disseminating product functionality*, and *increasing team cohesiveness*. The sharing of information in the review process reduces project risk associated with the loss of critical people. And additional emphasis on *traditional test activities* can help the inexperienced become familiar with the product, process, and team.

How Often the Product Will Be Modified

A product with an ongoing lifecycle of sequential product versions or updates can benefit greatly from a *reusable test infrastructure* and stronger emphasis on usable product documentation, gained through a more rigorous *review and change management approach*. The value of effort spent on these up-front activities increases as a function of the overall expected life of the artifacts generated. The converse of this is that for one-shot projects such as proof-of-concept prototypes, there is limited value in these activities beyond specifying and exercising exactly what the prototype is being developed to confirm or reveal.

How Often the Product Will Be Tested

Short test cycles increase the need for an *automated suite of tests* that will drastically reduce the costs of running comprehensive regression tests (and hence increase the likelihood that they will actually be run). If the product is being managed with frequent builds (often daily or more frequently), an automated *smoke test* can confirm the soundness of the build and thus avoid the disruption caused by releasing garbage to the development and test teams. If automated GUI-based test packages are used, it is often necessary to enforce a *GUI freeze* into the development lifecycle to avoid the unnecessary frequent revision of test scripts. If automated tests are based on comparison of results against a "golden set" of test results, *thorough inspections* of this golden set are important to ensure that you are not continuously generating erroneous test results.

One of the primary reasons that unit tests are rarely performed is that this requires generating additional code to execute an isolated portion of code and to provide appropriate stimulus to the software under test. Organizations commonly view the additional code (along with documentation, etc.) as "project overhead" and believe it will tend to delay a project. This infrastructure, in the form of low-level stubs that can sit below the software under test and scaffolding that can sit on top and allow control of inputs and response, can often account for as much code as the application itself or more. The infrastructure's value, however, lies in the ability to exercise code to a more detailed level, to rerun tests and perform them in an automated fashion (which supports regression tests as the software evolves over time), and to provide early insight into the quality of the artifacts generated.

For software projects, where there is a successive building upon previous elements (both across phases, from requirements to design to code, as well as in phase, from foundational software elements

©2002 Cutter Information LLC

to those higher-level elements that depend on them), the elements developed earlier will have an inherently higher impact overall. One could also argue that in the earlier phases of a project there is greater uncertainty, and hence there is a greater probability that defects will be injected.

Organizations should give priority to the up-front and high-leverage activities that will provide greatest value to the project.

CONCLUSIONS

Prioritizing testing activities is important, and organizations should address it in several ways. They must consider staff availability, overall coverage as driven through risk mitigation strategies, balance and interaction of the testers and developers on the project, and the skill sets of those testers and developers. Organizations should give priority to the up-front and high-leverage activities that will provide greatest value to the project. As these strategies are embraced from project to project, however, it is important to avoid an all-out change in test strategy to reduce the risk of culture shock and project failure. Chew off a bit more than you are doing now in the planning and testing areas, and you are more likely to realize the benefits and increase your chances of success. In turn, this will allow you to achieve the cultural acceptance that will facilitate further improvement inroads downstream.

With appropriate planning of test activities, proper prioritization will result in several significant benefits that will improve product quality dramatically. Up-front activities will allow you to smooth resource utilization on a project, bring the development and test teams together sooner for more effective collaboration, and reduce overall project risk through earlier verification of project artifacts.

REFERENCES

1. Boehm, Barry, and Victor Basili. "Software Defect Reduction Top 10 List." *IEEE Computer*, Vol. 34, No. 1 (January 2001), pp. 135-137.

2. DeMarco, Tom. *Controlling Software Projects*. Prentice Hall, 1998.

3. Hanna, Magdy. "Disciplined Software Testing Practices." Presentation at the *Sixth International Conference on Practical Software Quality Techniques (PSQT) & Practical Software Testing Techniques (PSTT) 2000 North*, Minneapolis, Minnesota, USA, September 2000.

4. McConnell, Steve. *Software Project Survival Guide*. Microsoft Press, 1998.

5. O'Neill, Don. "National Software Quality Experiment: Results 1992-1999." Presented at the *Software Technology Conference*, Salt Lake City, Utah, USA, 1995, 1996, and 2000.

Jim Brosseau has 20 years' experience in the software industry in a wide variety of roles, application platforms, and domains. A common thread through his experience has been a drive to find a less painful approach to software development. Mr. Brosseau has worked in quality assurance at Canadian Marconi and was involved in the development and management of the test infrastructure used to support the Canadian Automated Air Traffic System. He is principal of the Clarrus Consulting Group in Vancouver, Canada, and in the past four years he has consulted with numerous organizations throughout North America, specifically to improve their development practices.

Mr. Brosseau is the editor of the Software Productivity Center's E-ssentials *newsletter (www.spc.ca/essentials/) and has authored several of the lead articles. He has been published in* PM Network *magazine, the PMI GovSIG newsletter, and the* SEA Software Journal*, and he has made presentations to* Comdex West, PSQT North*, the New Brunswick SPIN group, and several local associations.*

Mr. Brosseau can be reached at Clarrus Consulting Group Inc., 7770 Elford Avenue, Burnaby, BC V3N 4B7, Canada. Tel: +1 604 540 6718; Fax: +1 604 648 9534; E-mail: jim.brosseau@clarrus.com.

Cutter IT Journal

Topic Index

August 2002	Plotting a Testing Course in the IT Universe
July 2002	Confronting Complexity: Contemporary Software Testing
June 2002	B2B Collaboration: Where to Start?
May 2002	Information Security and Privacy in a Fragile World
April 2002	Web Services: "You Say You Got a Real Solution..."
March 2002	The Technology Myth in Knowledge Management
February 2002	Is Risk Management Going the Way of Disco?
January 2002	The Great Methodologies Debate: Part II
December 2001	The Great Methodologies Debate: Part I
November 2001	BI and CRM: Critical Success Factors for Achieving Customer Intimacy
October 2001	The Future of SPI
September 2001	Testing E-Business Applications
August 2001	Enterprise Application Integration

CUTTER IT JOURNAL

Upcoming

Issue Themes

Design for Globalization

XP and Culture Change in an Organization

Mobile Wireless

Preventing IT Burnout

Open Source

http://www.cutter.com/ or +1 800 964 5118