An Introduction to Customer Focused Test Design

Alan Page

alanpa@microsoft.com

Abstract

Test design, simply put, is what testers do. We design and execute tests to understand the software we're testing, and to identify risks and issues in that software. Good test design comes from skilled testers using a toolbox of test ideas drawn from presentations, articles, books, and hands-on experience with test design. Great testers have great test designs because they have a generous test design toolbox.

One significant drawback of the majority of test design ideas used by many testers is the heavy emphasis on functional testing of the software. While functional testing is a critical aspect of software testing, many test design ideas fail to include high-priority plans for testing areas such as performance, reliability, or security. Test teams frequently delegate these testing areas to *specialists*, and ignore the importance of early testing. Furthermore, when testers manage to design tests for these areas, the testing often occurs late in the product cycle, when it may be too late to fix these types of bugs.

Our team at Microsoft has introduced the concept of Customer Focused Test Design. This test design approach includes an emphasis on testing end-to-end scenarios, real-time customer feedback, future customer trends, and, most importantly, a shift of emphasis away from functional tests, and towards early testing approaches for quality attributes that have the biggest influence on the customer's perception of quality.

This paper discusses the individual pieces of this approach, the success we've had so far, and provides examples of how this change in approach has affected the overall project outcome.

Biography

Alan Page began his career as a tester in 1993. He joined Microsoft in 1995, and is currently a Principal SDET on the Office Lync team. In his career at Microsoft, Alan has worked on various versions of Windows, Internet Explorer, Windows CE, and has functioned as Microsoft's Director of Test Excellence. Alan is a frequent speaker at industry testing and software engineering conferences, a board member of the Seattle Area Software Quality Assurance Group (SASQAG), and occasionally publishes articles on testing and quality in testing and software engineering web sites and magazines. Alan writes about testing on his blog (http://angryweasel.com/blog), was the lead author on How We Test Software at Microsoft (Microsoft Press, 2008), and contributed a chapter on large-scale test automation to Beautiful Testing (O'Reilly Press, 2009).

1. Introduction

Test Design is what testers do – but are we designing and running the best set of tests to ensure that customers desire and crave our products? Our team at Microsoft is exploring a set of alternate approaches to test design with the hope of designing tests that discover issues that affect the customer perception of quality.

Testers design tests, perform them, and make decisions based on those results: Are we done? Do we run more tests? Do we have new test ideas? This is the essence of test design and a large part of what testers do. Lee Copeland's book on test design is one of my favorites on this subject.[1] It's a wonderful book on functional testing, and should be part of every tester's library. However, like many other books on test design, its primary focus is on functional test design, such as equivalence classes and boundary values. Even exploratory testing approaches, depending, of course, on the tester and the testing mission, often drift more towards functional testing than customer-focused testing. Today's test design approaches and techniques are effective, but are they good enough for tomorrow's software? Can we do better?

A good portion of software testing today aims at verifying functionality, and there definitely *is* value in functional testing. Functional attributes are frequently the easiest things to spell out in requirements, and often some of the easier things to verify. Functionality is undoubtedly important, but we may be putting too much of our effort into functional verification while sacrificing the creation of highly usable and desirable software. I believe that testers frequently overemphasize functional testing to the detriment of customer focused testing. No one wants to develop a full-featured and perfectly functioning software program that disappoints or infuriates customers due to issues in performance, reliability, compatibility, or other areas that influence the customer's perception of software quality.

Today, customers demand software that does *more* than just work. Customers want software that is quick, reliable, secure, private, compatible, and usable. Most testers are aware of the importance of these "non-functional" types of testing, but most testing in these areas ends up delegated to the end of the product cycle, to test specialists, or to the customers themselves.

I often hear testers describe themselves as the "voice of the customer." It's critical to remember that while testers can be customer advocates, *we are not the customer*. We are contributing members of the product team and can only be an empathetic stand-in for what customers, providing some insight about how they might want the software to work.

Customer Focused Test Design starts as *testing with the customer in mind*, and contains several aspects that ensure an emphasis on testing for those issues that have the most impact on the customer perception of quality. This paper will discuss four of those aspects: Scenarios, Outside-In Testing, Live Testing, and Shifts & Sparks..

2. Emphasis on Scenarios

One approach to improving customer perceived quality is to build a product around large end-to-end scenarios. These large scenarios are a consolidation of several smaller scenarios or use cases that describe a primary workflow of the product. A project may only have a few of these –usually one or two for each major feature area. Early in the product cycle, scenarios give the whole team a clear idea of the product they're building - and how the product is going to help customers accomplish their tasks.

In their initial form, these scenarios tell a narrative story using the customer's voice. Ideally, they focus on the customer *experience,* not the implementation, and tap into emotion or reveal insight. As the product matures, these scenarios may be adapted to refer to specific product features. An example of a scenario of this type is:

Tina is a **fast-rising** sales manager at a Fortune 500 company. She manages a team of eight people who work at several sites across North America, Europe, and India. In order to manage her team and help them meet and exceed their sales quotas, she **wants** to be able to connect with her team frequently. On most occasions, a short message to her team is adequate, but when context dictates, she often **needs** to speak or see employees who are not co-located with her - not being able to efficiently connect with her team when needed is **frustrating**. She knows that teamwork is critical in order for her whole team to be successful, and she wants to build a culture of sharing and teamwork in order to enable success

We use these scenarios as a guide or charter for focused testing sessions. Because the scenarios are feature agnostic (meaning they describe the task the user wants to accomplish rather than the features they will use), testing tends to focus on ensuring the success of executing the workflow rather than on the testing the functionality of the feature. This also allows for variation in *how* a tester moves through the workflow, giving them a better chance of emulating customer usage patterns.

At this time, our team is at a relatively early stage of the product cycle, but we are already using scenarios as one of the primary progress and status indicators for the state of the product. Low-level metrics like pass rates, code coverage, or bug counts are details that can indicate risk or support a higher-level view of product status. However, we use the scenarios to verify that the product can accomplish customer workflows. It's good to have low-level metrics in case someone needs the details, but scenario status typically gives a much better overview of whether the software will fit the needs of customers. A classic method of reporting scenario status is a standard red-yellow-green report; where green indicates that the scenario is working perfectly, red indicates a scenario fails to complete or is blocked, and yellow indicates some level of intermediate status (see Figure 1). The decisions and details that go into a red-yellow-green report are beyond the scope of this paper, but are worth studying for those interested in methods of reporting project status.

Scenario 1 (green)	**Scenario 2 (yellow)**	**Scenario 3 (red)**
Scenario 4 (yellow)	**Scenario 5 (red)**	**Scenario 6 (green)**

Figure 1 - Example Red - Yellow - Green reporting for Scenarios

These scenarios are a prediction of what customers will do with the software, so the basis of the scenario must come from real customer research and studies and not simply a made up story. You can't predict everything, but basing your scenarios on as much real data as you have will give you a better chance of building the right thing the first time.

3. Outside-In Testing

Many testing reference books such as Jorgensen's "Software Testing"[2] refer to *Levels of Testing* where "testing" starts at unit testing and progresses through functional, system, integration, acceptance testing, and finally into non-functional areas such as performance, reliability, and usability.

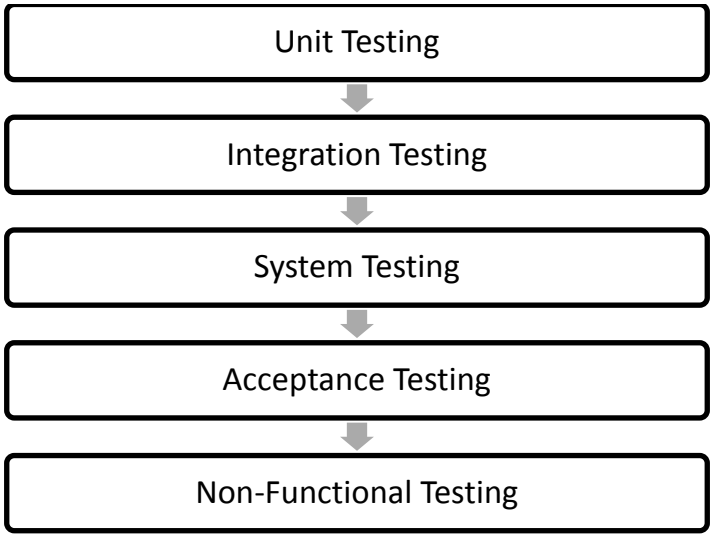

Figure 2 - Software Testing Levels

Typically, testers begin at the level where programmers stop testing. For example, if programmers write unit tests, testers typically start their effort with integration testing. If your programmers do **not** write unit tests, those basic tests may be the beginning of the testing effort.

The testing effort continues through the levels of testing, expanding the scope of testing to scenarios that cover wider areas of the product and towards scenarios with more customer facing attributes. For example, integration testing covers a wider area of the product than unit testing covers, and is closer in what it covers to a customer scenario. The non-functional areas generally cover a variety of scenarios and are a reasonable proxy for how customers will use the software. In this model, the final stage of testing is non-functional testing (e.g. performance testing). Unfortunately, the types of bugs found at this stage are often design issues or other deep-seated issues that nobody can safely address so late in the product cycle.

Most of us have seen or referred to this cost of change curve (Figure 3). While I agree that the cost of finding a bug later in the product cycle is more expensive than finding it earlier, it's important to consider that not all bugs are equal.

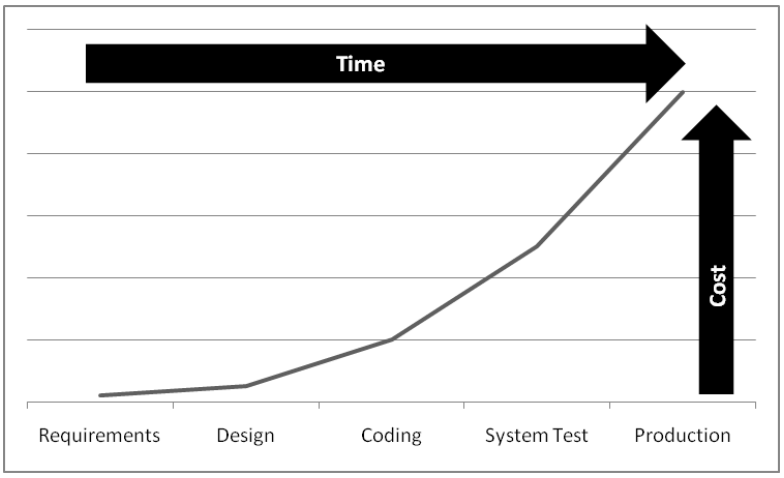

Figure 3 – Traditional cost of change curve[3]

In fact, the cost of change for a performance or reliability bug that occurs due to a design issue could have a much *steeper* curve (Figure 4).

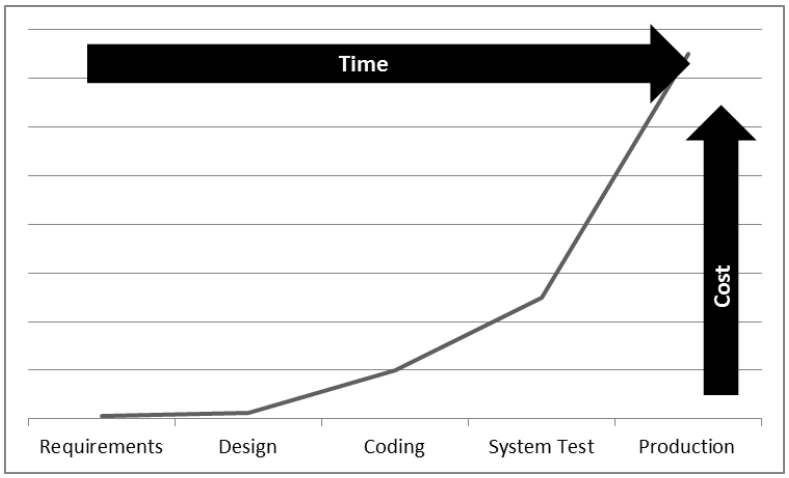

Figure 4 - Cost of change for design issues (hypothetical)

Whereas a functional bug – perhaps an off by one error[4], or a missing hotkey *is only incrementally more expensive to fix late in the product cycle* – often the only increase in cost is the potential extra time it takes to find the offending line of code (Figure 5).

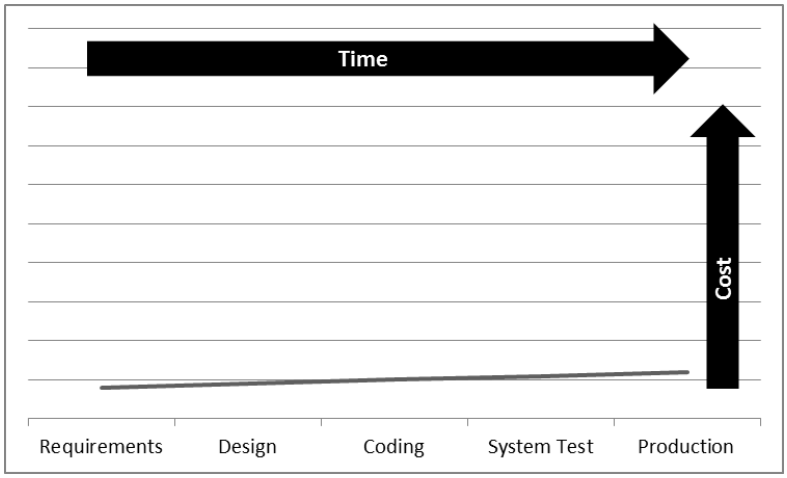

Figure 5 – Cost of change curve for functional bugs (proposed)

The solution we use to address this is an approach I call Outside-In testing (this name comes from a 1973 paper by Allan Scherr where he called the levels of testing "Inside-Out Testing"[5]).

The premise of the Outside-In approach is to put a substantial emphasis on non-functional testing early in the product cycle to uncover as many issues as we can in the areas of performance, reliability, usability, compatibility, security, privacy, and world readiness *before* we verify large parts of product functionality. Instead of progressing through these testing levels from top to bottom, we move to the bottom as soon as possible and work our way backwards (Figure 6).

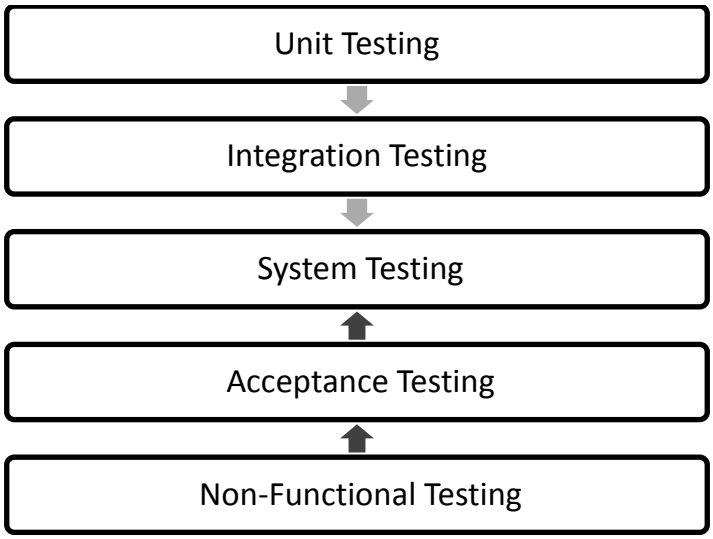

Figure 6 – Outside-In testing

There are two important aspects of this approach. The first is that we do not abandon functionality testing entirely. We've discovered that the product needs a level of functional quality before we can perform non-functional testing, so some early functional verification is necessary. The inverse of this is also true. To perform most non-functional testing, one must test the underlying functionality. For example, if I were to run a "stress test" of Windows Notepad, I may paste a large number of characters into the edit field, save

the file, edit the file, and then repeat the steps. The only thing that adds *stress* to the test is the size of the data. Each of the other steps is a simple functional test.

To ensure that every tester thinks about these non-functional test ideas early in test design, our test design template includes guidelines and requirements for designing non-functional tests at the initial stages of creating tests. Appendix A has an excerpt from this section of our template.

Finally, as with Scenarios, we prefer to report status metrics at a higher level than tests run and bugs found. We look at each of the key non-functional areas and report red / yellow / green status based on the count and severity of the bugs found in those areas.

Table 1 - Example non-functional scorecard

Area	Performance	Reliability	Usability	Security
Status	***Red***	*Yellow*	Green	Green
Bug Numbers	134, 137, 199	111, 174		

.

4. Live Testing

A stream of customer data is critical to react to problems and understand usage patterns. Microsoft collects information on application crashes via Windows Error Reporting[6] and collects considerable amounts of usage data through the Customer Experience Improvement Program.[7] These programs are extremely useful and provide actionable data that we use to improve our products. One drawback of this data is that we generally use the information to improve the *next* version of the product. While we gather this information during public betas, much of it still comes too late to have a significant impact on the current product.

One way of obtaining customer feedback quickly is through Testing in Production (TiP), and the approach is already popular on the teams at Microsoft that build web services, as well as at companies like eBay and Amazon. In the past, web service teams have attempted to build extensive pre-production environments to mimic real-world production sites. Unfortunately, most teams discover it's nearly impossible to replicate production environments and workloads; many issues are uncovered only in the production environment - where the combination of server hardware, scale, and real customer workloads reveal bugs that seldom show up on pre-production systems. The key to a successful TiP approach is extensive monitoring systems that can detect issues that affect customers immediately, resulting in immediate roll back. This enables teams to run tests against production environments and recover nearly instantaneously should an error cause a problem that may result in a customer issue.

We use TiP extensively for server and web components of Microsoft Lync. For example, we run a portion of our dogfood[8] program, as well as a large set of our test automation against production servers that serve real customers. The monitoring and diagnostics systems on our production servers allow us to identify when a problem occurs and roll back or disable tests, as appropriate, to minimize (or eliminate) customer impact.

For desktop client applications, where TiP approaches are not generally applied, we are making a deliberate effort to take some of the best parts of TiP and see how we can apply them to desktop applications. We have the advantage of building a product that the entire company uses for telephony and collaboration, which means that a beta program of twenty to forty *thousand* users is reasonable, even for early adoption.

Additionally, we are attempting to replicate some aspects of TiP with our rich desktop client. Although we can't update these early adopters in real-time, as we can with web services, we strive to gather their

feedback, make adjustments and get them a new build as frequently as possible. Our core team upgrades daily, and we attempt to upgrade many of our dogfood users outside of the core team once or twice weekly. Through customer usage data, we can tell, for example, which parts of the product get the most use as well as which areas get little use. This data prioritizes test investment and contributes to improved risk analysis. For example, if we discover that most Instant Messaging sessions last less than ten minutes and that most messages are between 20-80 characters, we can prioritize functional and non-functional test cases around that data, rather than focus on long strings and extended messaging sessions.

Another example of where we leverage TiP for our client application is A/B[9] testing. A/B testing, where customers see one of two different experiences, is a staple of online services. We can deploy two different versions of Lync and track usage data to help us make decisions about user interface design, usability, and many other customer-facing features. While not directly connected with how we design tests, the test team is responsible for implementing and monitoring A/B testing.

Our goal is to gather customer data and react to that data as quickly as possible, and we are continuing to investigate approaches borrowed from TiP to help us reach this goal.

5. Shifts and Sparks

Our team has pioneered another testing technique that we call Shifts and Sparks. The Shifts come from an initiative done by the Envisioning Lab at Microsoft.[10] The Envisioning Lab, part of Microsoft Office Labs, has performed extensive market research to develop a list of macro forces of change (Shifts) they expect to occur in the social, technological, economic, environmental, and political landscape over the next five to ten years.

For example, one of the shifts is the Content Revolution that anticipates the volume, diversity, and pervasiveness of digital content merging into communication and entertainment. Another shift is Mobile Computing that proposes mobile phones and platforms may surpass PCs to become the dominant personal computing platform.

Our test team uses the Shifts to help *Spark* new test ideas. Testers use a set of cards with Shifts (from the envisioning team) written on one side and Sparks (our test ideas) written on the other. As part of the test design process, testers scan through the Shifts and pick 2-3 that may apply to the area they're testing. Then, they read the Sparks and add tests or test ideas as applicable.

For example, in the area of Mobile Computing, the Sparks include concepts like "Always-on connectivity," and "Bandwidth Dependent Experience." If a tester is testing the Instant Messaging feature of Microsoft Lync, they may read the Spark and ponder the anticipated user experience should network connectivity or bandwidth change during the session. Because of the Spark, they may decide to test on a throttled network, or simulate bandwidth and connection issues to ensure a good customer experience.

Shifts and Sparks are possibly more applicable when reviewing specifications. The team that developed Shifts and Sparks is presenting a poster paper at the PNSQC 2011 conference with additional details on their process and findings.

The goal of Shifts and Sparks is to help testers come up with additional customer scenarios that may not be clear from specifications, code, or implementations. The Sparks help to inspire new test ideas that testers may not have considered using other approaches. These additional scenarios contribute to and enhance the other forms of scenario and end-to-end tests performed by the test team.

6. Next Steps

Our Customer Focused Testing approach is still in a relatively early phase; however, we have already observed several changes that we believe are due to the shift in test design focus. One of the most notable changes is the increase in customer focused thinking across the team. Achieving customer focus was the intent of the effort, but the culture change has been noteworthy. Not only does the team have a customer focus, there is also a "buzz" about relating to the customer. In addition, the non-test disciplines have an increased awareness of customer facing issues due to the increased emphasis on the subject from the test team.

We've received positive feedback from management and peer disciplines on each of the approaches discussed in this paper. Most notably, our program management staff has begun to incorporate shifts and sparks into their planning process. The test team still uses the sparks to generate new test ideas, but our program management team is using the same approach to improve and enhance functional specifications.

One challenge in our approach is appropriately conveying the de-emphasis of functional testing. Because of the focus on scenario-based testing, it has been challenging to reeducate the team to expect some functional bugs to surface in later stages of the test cycle, or to our internal dogfood users. We're slowly making progress with this approach and showing the value in discovering non-functional issues early. In a few cases, we're even seeing developers put more effort into writing unit *and* functional test cases to discover or prevent functional bugs in the first place. One of the original goals of the customer focused test design initiative was to promote developers owning unit and functional testing, leaving testers to focus on end-to-end and non-functional testing. Seeing this change occur, even in a small part of the team, demonstrates the merit of the approach.

Overall, we are pleased with the initial progress but know we have to carry it through to a shipping application before we can truly claim victory. However, we feel that the efforts so far will have a positive impact on post-release customer satisfaction, and believe this could be a useful approach for any test team.

Appendix A – Test Design Information

Customer Focused Testing

The following subsections contain the non-functional areas that have the highest impact on the customer perception of quality. Use this section to describe how you will test for these areas.

1. Security

It is important Microsoft customers can trust all our features to be secure. Costs to Microsoft in QFEs and patches, service downtime, credibility, and resulting potential loss of future business are serious.

2. Privacy

Are there tests in place that validate the protection of customer data? How does the software manage the collection, storage, and sharing of data? Are privacy choices properly administered for users, enterprise, parent components? Are all transmissions of sensitive data encrypted to avoid the inadvertent exposure of data? Does the application provide a mechanism to manage its privacy settings during the install, first-run experience, or before data is ever transmitted or processed?

3. Usability

What are the current usability (including accessibility) issues of the feature? What is your approach to identifying new usability issues? What is the usability goal and criteria for this feature?

If the feature is UI (i.e. is seen by an end user), you must include accessibility testing in this test plan. Include testing for high DPI settings, high contrast screen modes, screen orientation, do sounds have visual cues, is everything *sensibly* accessible via the keyboard?

4. Reliability

Discuss your strategy for testing feature reliability. This testing may also need to cover memory/thread leak detection and fault injection depending on the feature area. What are the expectations for reliability when tested under low-resources, unavailable-resources, and high-latency conditions? What is the strategy for long-haul testing of this feature?

5. Performance

Describe how the performance testing will happen. Testing includes simulating load conditions, validating correct operations, establishing the proper mix of operations, and generating the necessary supporting data and environment. Performance doesn't always mean fast – but ensure that performance meets guidelines, standards or expectations.

6. Compatibility / Interoperability

Compatibility is the ability for a program to work with other programs or plug-ins – including previous versions. Interoperability is the ability of the component parts of a system to operate successfully

together. Consider the potential for compatibility or interoperability issues in this feature. For example:

- Will this feature interact with client or server components [from the previous version]?
- Are there any MS or 3rd party add-ins for a previous version of Lync that interact with this feature?
- How does this feature interact across platforms or devices?

Discuss how you will address any issues above. Include a compatibility test matrix if necessary.

7. World Readiness

Discuss the globalization issues with this feature. Consider character issues in text fields as well as formats for date, time, currency, etc.

References

[1] Lee Copeland, *A Practitioner's Guide to Software Test Design*., 2004.
[2] Paul Jorgensen, *Software Testing: A Craftsman's Approach*, 1995; Glenford J. Myers, *The Art of Software Testing*. 1979.
[3] Barry Boehm, *Economics of Software Engineering*, 1981.
[4] An "off by one" error is a functional error that occurs when a program iterates either one too many, or one too few times through a loop. This is common in most programming languages due to zero-based counting and operator mis-matches.
[5] William Hetzell, *Program Test Methods*, 1973.
[6] Microsoft, "Windows Error Reporting," Microsoft, http://msdn.microsoft.com/en-us/library/bb513641.aspx (accessed August, 2011)
[7] Microsoft, "Microsoft Customer Experience Improvement Program ", Microsoft, 2009, http://www.microsoft.com/products/ceip
[8] *Dogfood* is a term used to describe a team that uses its in-progress software to get daily work done. *See* http://en.wikipedia.org/wiki/Eat_one's_own_dog_food
[9] A/B testing presents two (or more) user experiences to customers and tracks data (e.g. number of clicks on a target) for the control and experimental experiences. *See* http://en.wikipedia.org/wiki/A/B_testing
[10] Microsoft, "Envisioning," Microsoft Office Labs, http://www.officelabs.com/Pages/Envisioning.aspx (accessed August, 2011)