Are You In It For the Long Haul?

Brian Rogers

brian.rogers@microsoft.com

Abstract

Unit tests, regression suites, and end-to-end scenarios are invaluable in establishing a baseline measurement of software quality. They are also wholly insufficient for determining whether a product is truly ready to ship. To aid in this decision, we must regularly collect data on how well a system stands up to longer periods of use (and abuse). Long-haul testing is a methodology uniquely suited for this task.

This paper demystifies long-haul testing and shows how this valuable but often poorly understood technique can pay dividends if appropriately defined and implemented. Learn the principles and practices for building effective long-haul tests to simulate production usage patterns, leverage controlled chaos, and achieve high coverage while maintaining reasonable cost.

With ever-increasing user expectations around high availability and reliability, it is now more important than ever to adopt and embrace long-haul testing for your project. So are you satisfied with short-term quality indicators, or are you in it for the long haul?

Biography

Brian Rogers has been a software tester for over a decade. In his current role as a principal software development engineer in test in the Windows Azure group at Microsoft, Brian designs and develops tools and automated tests for large-scale distributed systems. Brian has a B.S. in Computer Engineering from the University of Washington.

1. Introduction

The software industry is now enlightened enough that it is no longer a question of *if* you test but rather *what* and *how* you test. Developers are test-infected[1] and unit testing is on the rise[2]. Test organizations at companies like Microsoft[3] and Google[4] boast a wealth of automated functional and integration suites which run against every build and protect against regressions. Such testing practices are generally treated like oxygen – essential for survival – and few would argue against their necessity in establishing a baseline software quality measurement.

However, today's software runs on devices like tablets and smartphones that may go for weeks without a restart. Cloud services and web applications operate in always-on mode and despite the unreliable nature of the underlying infrastructure, user expectations around availability and reliability are high[5]. Using simple test cases to exercise the software for mere seconds or minutes at a time is clearly not enough to determine if a product is ready to ship. A comprehensive software testing effort must evaluate longer periods of use with adequate coverage of faults and error paths.

When faced with this challenge, many test organizations turn to the relatively well-known practice of **stress testing**. While useful, stress by definition is meant to cover narrow, extreme situations. As it turns out, the lesser-known but related methodology of **long-haul testing** is a better choice for evaluating a broader set of typical application workloads.

2. Defining "long-haul"

Stress, performance, long-haul, and a variety of related non-functional test types can all be grouped under the blanket term **load testing**. The exact meaning of load testing differs slightly depending on the source but I use a working definition of "observing and evaluating a system at varying usage levels." Given the amount of confusion and conflation that exists with these terms, it is instructive to clearly delineate stress and long-haul testing to compare and contrast the two practices.

2.1 Stress testing

Stress testing is a specialization of load testing and is defined by IEEE as "testing conducted to evaluate a system or component at or beyond the limits of its specified requirements."[6] As such, failure of some sort is essentially the expected behavior in a stress test. Various subcategories of stress testing exist such as soak testing and spike testing; the main difference among all these test types is the way load is applied to the system under test.

In stress testing, the goal is to determine if the system degrades gracefully after hitting a breaking point and perhaps most importantly, if it *recovers* after restoring functionality or reducing load. For example, a stress test may intentionally overwhelm a server with so much user traffic that the system stops responding in a timely manner. After reducing the traffic, the system should bounce back and serve requests just as fast as before the stressful condition was introduced.

A good stress test is repeatable and sufficiently constrained in coverage such that it can reliably reach a known load target while exercising a small set of components. These qualities allow the test to establish a baseline, thereby providing data on unexpected changes in the defined operational limits of the software under test.

2.2 Long-haul testing

Long-haul testing seems to have no industry-standardized definition and only a handful of online references even mention it. (Arguably, the best overview is a write-up of how the Windows® CE team

used long-haul testing[7], wherein the author notably distinguishes the approach from stress testing.) I describe it as "observing the behavior of a system when given varying workloads over a long duration." A useful long-haul test runs for a minimum of several hours while subjecting the system to a variety of faults and actions (the "workloads"), typically concurrently. The workloads should bear some resemblance to realistic use cases, although they may be weighted to include more faults than typically seen in production to improve error path coverage.

Unlike stress, long-haul testing does *not* attempt to exceed known system limits; it has more in common with accelerated-life testing, where time and scale are compressed to allow more activity in a given interval than what is otherwise possible in a real-life, real-time scenario. A long-haul test passes if it observes no fatal errors (e.g. total system failure, data corruption) and if the system remains operational in the presence of any isolated or incidental faults, subject to previously determined service level agreements (SLAs).

The degree of concurrency, wider scope, and longer durations typical of long-haul tests make them at best semi-predictable (i.e. not *completely* random). Although it sounds less than ideal, this is in fact a benefit; given enough time to run, long-haul tests should uncover a broad range of unanticipated operational issues. As a consequence – and in sharp contrast to more narrowly focused stress tests – long-haul testing does *not* guarantee detection of specific regressions. (In some cases where more repeatability is desired, it is possible to use predetermined random number seeds or more deterministic workloads but these should be used with care to retain the overall benefits of long-haul.)

3. Why do we need long-haul testing?

By their nature, unit, functional, and other similar tests are **short-term quality indicators**. They provide quick feedback on whether a single component or small set of features works as intended when exercised in a reasonably controlled fashion – important and necessary, to be sure, but not *sufficient*. The fact is that real usage patterns for an application rarely involve a single walk through one or two features individually, but rather a diverse set of multi-step actions executed occasionally over hours or even days. For multi-user applications the interactions are even more complex, especially when taking into account the ways in which actions by distinct users (even when separated by long time intervals) can have combined or conflicting effects.

Classic integration testing techniques can help to gain *coverage* of these more complex scenarios but planning, writing, and executing individual test cases to exercise all such interactions is normally very difficult and cost-prohibitive. Add to this the need to cover the usual error paths – for cloud services, consider the annoying but unavoidable cases of network interruptions and backend server outages, to name a few – and the difficulty further increases. In fact, it is even more challenging since we must do all of this *repeatedly* for extended periods of time to move beyond the short-term and assess the longer-term operational behavior of the system.

Instead, we need a testing methodology that relies less on costly pre-planned workflows and embraces **controlled chaos**, that exercises the software under test with days' or weeks' worth of user activity but compresses time and scale to **minimize execution cost**, and that **stays within operational limits** in the process. In other words, **we need long-haul testing**.

3.1 The bug payoff

To further motivate long-haul as a uniquely valuable test technique, I present some actual product bugs I have witnessed as the result of long-haul testing:

- **The slow leak.** Objects allocated from certain operations were never destroyed resulting in slowly but steadily increasing memory usage. The issue was only noticeable if operations were performed repeatedly over a period of hours.

- **State poisoning.** A node transitioned into an undefined/unexpected state momentarily due to a software bug. Data in the system was replicated and so, if timed just right, the undefined state was saved to disk and transmitted to another node. When a node was restarted, it recovered its saved state from disk; however, the unexpected state caused a crash, preventing the node from ever starting again. This race condition required just the right mix of state changes and induced faults to observe – unlikely in a shorter, predictable test sequence, but hit relatively often in more randomized long-haul tests.

- **Too much information.** Tracing from certain components was overly verbose, leading to extremely large diagnostic trace files if the system ran continuously for an extended duration. The problem was not apparent until a customer workload test ran; the trace files were flooded with millions of repetitive trace lines which interfered with failure investigations.

Even with a full range of other tests (unit, functional, integration) already in place, these defects and many others like them can languish undetected.

3.2 A note about applicability

What about systems which do not have multi-user state, which have limited concurrency, or which do not need to operate continuously for long durations? As it turns out, even in these cases there is still some value in applying long-haul principles to improve coverage and reduce testing costs. Applying semi-random, lightly concurrent, time-compressed workloads and corresponding faults to the system under test can be a relatively low-cost way to ensure broad coverage of a variety of positive and negative code paths.

4. Can vs. should: goals and non-goals

When discussing the goals of long-haul testing, the mantra “just because you *can* doesn’t mean you *should*” is apt. Long-haul tests are invaluable but they are far from the only tests you will ever need.

4.1 Goal: reduce the cost of testing

Unlike comparable functional or integration tests, long-haul tests are not highly orchestrated or strictly prearranged, a definite win in terms of the cost of test planning. The longer running time combined with semi-random actions and faults give long-haul tests an edge in terms of cost vs. benefit – overall, you need fewer tests to cover a wider variety of product code paths.

4.2 Non-goal: supplant functional testing

Long-haul tests are powerful but not omnipotent. Functional testing is still the best way to get guaranteed regression coverage or to evaluate specific product use cases. Twisting a long-haul test to be 100% deterministic degrades quickly into a “worst of both worlds” situation; such a test would not only run for a long time but it would cover only a fraction of possible code paths.

4.3 Goal: uncover race conditions and invalid states

Race conditions occur when shared data in a program is modified by multiple threads without proper synchronization[8]. These and other concurrency issues are notoriously hard to debug as well as hard to find through traditional testing techniques[9]. While no test will uncover race conditions in a reliable and repeatable way, long-haul testing is most likely to catch broad concurrency issues throughout a system. The basic recipe for a long-haul test is “randomness + parallelism + time” – exactly what is needed to reproduce a typical race condition.

An invalid program state occurs when an invariant is violated, often codified by an "assert" in the source code[10]. Long-haul tests, given their breadth, tend to cover a large number of program execution paths while also building up a fair amount of previous state given their longer running time. They are therefore quite good at unwittingly running into situations which "should never happen" according to the product design or specification. When combined with a "fail fast" methodology (i.e. the program terminates as soon as an invalid state is detected[11]), a long-haul test is an inexpensive way to find invalid state defects; simply run the test and generate one bug report every time a new "assertion failed" crash fault occurs.

4.4 Non-goal: exhaustively validate all behavior

Automated testing of any kind must use oracles to make a pass/fail decision[12]. For a directed functional or integration test, the oracle is usually just a simple check against a predetermined expectation, e.g. "when function F is called, result R should be returned." For a long-haul test, things are not as simple. With a relatively chaotic mix of actions and faults, certain failures are expected and thus should be tolerated at least some of the time. It is impractical to devise a true oracle to cover all possibilities that the test may encounter. Most long-haul tests instead use a much simpler heuristic oracle[13] which explicitly rules out exhaustive behavioral verification of the system under test.

4.5 Goal: provide valuable feedback for ship-readiness

A well-designed long-haul test should make an important statement about product quality. The end result is one aspect, which is a testament to the reliability of the product ("the long-haul test passed after subjecting the product to 24 hours of continuous user activity and faults"). In addition, the test can provide operational statistics which help validate or challenge assumptions about system capabilities ("the product fully recovered from all network faults but the observed downtime was longer than two minutes in 50% of occurrences").

4.6 Non-goal: provide quick feedback

A "quick long-haul test" is an oxymoron. Reducing the running time of the test will have the likely result of providing less coverage since less state will be built up and fewer actions will be executed. Many other types of tests exist which can provide quick feedback if required.

4.7 Goal: leverage controlled chaos

Chaos is a virtue in long-haul testing when it is *controlled* – instead of "*any*thing goes," think "*many* things go." This is the key that allows a long-haul test to have broad enough coverage but with some reasonable expectations around what can be considered correct or incorrect behavior. Randomness should be leveraged judiciously and avoided when its use would add extreme or unnecessary complications.

As an example, consider a test that needs to generate files which must be later validated for data integrity. Generating completely random files could significantly complicate the data validation to ensure the files are uncorrupted. Instead, the test could produce a file using a specific pseudorandom number generator and a randomly generated seed; the files created by this test would be random for all intents and purposes but could always be checked for complete consistency given the original seed.

4.8 Non-goal: "monkey" testing

It is trivial to write a "monkey" test, i.e. one that executes random actions at random times for random durations; consequently, it is nearly impossible to say anything of value about the results of such a test. It is hard to even understand what the test will do until it is run and then what actual use cases its activity

maps to. These are problems for both the tester who must analyze and interpret the results and the project stakeholder who must decide on the merit and relative priority of a proposed defect.

5. One category, many flavors

There is not just a single type of long-haul test but rather a large category of tests that come in all shapes and sizes. The specific “flavors” of long-haul, as I refer to them, will differ depending on the needs of the project. The following are some examples of the types of long-haul tests that I have used on previous projects. This is not an exhaustive or definitive list but rather a starting point for practical consideration.

5.1 Low-level

A low-level long-haul test typically runs a stripped down version of the product in a “one-box” test environment, where a single machine hosts all necessary components. These tests are often written close to the code (like unit tests) to take advantage of internal components and data structures. For this reason, low-level long-haul tests are useful when there is a need to manipulate or observe internal state, e.g. to do more detailed validation or to inject low-level faults.

The more local nature of these tests makes them an attractive option in terms of ease of execution and debugging, especially when compared to a fully distributed test of the production system. However, to achieve this locality, many components may be simulated or absent thus removing them from the purview of the test.

5.2 Feature/subsystem

A feature or subsystem long-haul test focuses on functionality relevant to a specific area of the product. This focus necessarily constrains the surface area of the test and keeps it from becoming overly complicated or hard to investigate while still maintaining broad coverage of the area in question. This ensures a favorable cost/benefit ratio but scopes out integration coverage across multiple areas.

A feedback loop can exist between feature long-haul tests and functional tests in the same area. Functional tests can drive requirements upward to long-haul tests around coverage enhancements. Long-haul tests can push data downward about issues that warrant regression coverage or risk areas that require additional functional validation.

5.3 Customer workload

A customer workload long-haul test is a translation of a customer’s actual use cases into a test scenario, almost like a long-running acceptance test. Occasionally, real artifacts such as data files from the customer are used in the test workload. This is not always possible or desirable so these artifacts are often simulated or approximated. I have had success here by meeting with key contacts from the customer team (usually developers and testers) and asking detailed questions about their environment and usage patterns; I would then translate these into test workloads using my own test infrastructure so that I would retain control over the design and implementation of these artifacts. The drawback is that these simulated artifacts would require some maintenance over the lifetime of the project as scenarios change; however, the cost is usually lower than trying to fit real customer code and data into controlled test environments.

Customer workload long-haul tests are far more specific than most other long-haul tests and may only cover a few particular patterns of behavior. However, these tests tend to span many product subsystems, making them higher level relatives to integration tests. They are best employed in cases where a

customer uses the product in an otherwise uncommon configuration which has less overall representation in other test areas (e.g. the use of a unique or specialized network topology).

5.4 Full-system

A full-system long-haul test operates at the highest level, integrating multiple areas together and exercising cross-component workloads and system-level faults. These tests are usually the most difficult to develop and analyze given the number of possible sources of errors and defects. However, for the same reason they are also likely to uncover issues that no other test would be able to find.

A feedback loop can also exist here between full-system and feature/subsystem long-haul tests. Defects found at this level that exist within a single area may be pushed down to the appropriate feature long-haul test, while known missing integration coverage can be pushed upward to the full-system long-haul test.

6. Automation design considerations

The eventual success or failure of automated long-haul testing will be due in large part to the design decisions made early in the test lifecycle. Based on my past experience, I present the following considerations for designing effective long-haul tests.

6.1 Define the basic test architecture and topology

Before the test implementation phase, it is wise to decide on what a long-haul test should look like. There are many important decision points as far as the details go, but most tests boil down to the following basic description: a long-haul test **drives concurrent** and **continuous** workloads made up of **actions** which exercise product functionality and **faults** which force **recoverable** errors, subject to **periodic validations** to decide whether the system is operating correctly.

6.1.1 Actions

Depending on the context, an action can be a function, a class or interface implementation, or even a separate executable. The design for actions in turn pushes requirements for the test driver, which in the simplest case can be a loop with a random number generator or in a more complex system, a distributed work scheduler. Where the actions run is dictated by the topology or physical layout of the test which also drives decisions on the usage of parallel threads, separate processes, and remote machines.

6.1.2 Concurrency

The level of activity must be bounded to avoid exceeding operational limits but should be high enough to keep the system busy. It is simplest to provide a predetermined “safe” upper bound to limit the amount of work (e.g. “ensure a maximum of 100 concurrent requests”). It is also possible to use an adaptive design which discovers limits dynamically but this is somewhat difficult in practice to implement, and even more so in a fully distributed topology.

6.1.3 Faults

Fault injection is a massive topic in itself and a thorough treatment would require pages of exposition. Instead, I will summarize the key points as they apply to long-haul tests.

Higher level tests should focus mainly on external faults, i.e. those coming from *outside* the system. Process crashes, network outages, and disk errors are typically easier to inject at a high level and better represent the faults that occur in production. Lower level tests which are closer to the code can opt for a

more intrusive internal fault injection strategy, e.g. causing internal functions to throw exceptions or forcing memory allocators to fail.

Ensure that any given fault has a defined recovery action, whether automatically taken by the system or explicitly invoked by the test. It does not do much good to have a long-haul test that intentionally forces the system into a terminal error state and then simply stops. (Of course, if the system is designed to safely recover from the fault but does not, this is an obvious product defect.)

6.1.4 Validations

It is important to clearly specify the set of validations which are applicable to a given long-haul test. These ultimately determine the expected results and the success criteria for the test. Identify the data needed to validate a given set of actions and decide how to collect it. Note that some validations may be able to operate against partial state or incomplete knowledge, while others require the system to "pause" while they gather and check the complete state. Understand the implications of the validation strategy you choose and ensure the product supports it – if not, this could be a sign of a product testability or design issue requiring further discussion within the team.

6.2 Separate actions from validations

Tightly coupled actions and validations are rarely a good idea even in functional tests as they limit reuse and lead to poor test design. This approach simply does not work for long-haul tests where the same actions can be performed at different times under different conditions with different expected results.

Instead, think of actions as data producers and validations as data consumers. An action may produce multiple data items, e.g. request timestamps, the time taken to respond, the error code or codes, and the user state associated with the requests. Distinct validations can then be designed which consume all the relevant data items for the checks being performed with no need to care about which action (or actions) produced them.

6.3 Parameterize test inputs

It is a good idea in general to allow configurability of test behavior without having to change the code. Long-haul tests given their larger scope and longer durations tend to require even more tuning over their lifetime than other more targeted tests. Parameterize early and often, and not just for data values – consider allowing selection of different types of test actions and validations if appropriate.

6.4 Use profiles to guide decisions

Consider a software product which maintains online backups of user data. The full test matrix for such a system could be huge – there are essentially infinite possibilities for how to choose the file sizes, the file contents, the rate of uploads and downloads, and so on. An effective way to turn this into a finite test matrix is to consider user profiles. Use production data if you have it or market research and projections to decide what users are actually doing with your software. Perhaps you can determine that there are three basic profiles that most users can be grouped within, each associated with some general usage patterns (e.g. a "casual user" uploads about 100 files per day, 10 MB per hour, etc.).

6.5 Define partitions for disparate test actors

Validation of a long-haul test is often difficult, but the task can be simplified by partitioning the various test actors and workloads. Again, using the online backup example, imagine that users can create shared folders which get synchronized back and forth between multiple machines. Throwing multiple concurrent workloads at the system to indiscriminately add and remove files from a shared folder will quickly become

hard to manage and verify. Instead, you can use the shared folders as partitions to group different user workloads. One shared folder could be associated with a set of users that read and write only distinct files with an expected result that no conflicts should be observed. Another shared folder could be associated with users that read and write the exact same group of files with the expected result that conflicts are occasionally observed. This partitioning allows more detailed validations and coverage of still semi-random but better targeted use cases.

6.6 Coordinate invasive faults

In some cases, relatively isolated faults if injected in quick succession or at inopportune moments will cause undue pressure on the system. For example, imagine a fault that power cycles a single backend server. If the system is designed to take up to one minute to recover from this fault yet the test injects these faults once *per second*, the system will quickly lose all backend servers. These kinds of faults should be properly scheduled and coordinated to avoid turning a long-haul test into a stress test.

6.7 Optimize for diagnosability

Long-haul testing will never guarantee reproducibility; instead, strive for **diagnosability**. Every long-haul test should create a complete and sufficiently detailed log of all actions performed to aid in root cause analysis of any discovered defect. (It goes without saying that this is not a test-only responsibility – the product itself must also have enough instrumentation and logging of its own actions.) Be aware that it is definitely possible to run into log size issues given the long duration of the tests; to avoid running out of disk space or creating files which are too hard to manage, consider using segmented or circular log files for both test and product components.

7. What success looks like: project planning and exit criteria

The decision to adopt long-haul testing is only the first step. To actually make this test effort a success, the team as a whole must come together and commit to seeing it through to the end of the project. The best long-haul initiatives use a **common vocabulary**, clearly define the **scope and target** of the tests, align the work to a **realistic schedule**, and **iterate** to reach a desired end goal.

7.1 Use a common vocabulary

Make sure that your organization understands and uses the same terms to describe the long-haul test effort. In particular, be ready to compare and contrast any similar load testing efforts that your team may already be doing. Although it sounds obvious, a common vocabulary is essential and simple misunderstandings can easily derail an otherwise solid plan.

7.2 Agree on the scope and target

Will you use customer workload tests, feature tests, or a mix of long-haul test flavors? Which behaviors will you focus on and what validations will you apply? Are any areas of the product specifically excluded or out of scope for the test effort? Perhaps most critically, *how long will these tests be run* and *what are the expected results*? A successful long-haul initiative requires clear answers for all these questions and likely many more.

Strive for organizational agreement as much as possible when deciding on the answers. While there are some decisions more naturally left to the discretion of the test organization (e.g. test planning, design and architecture), others must involve partner engineering disciplines and project stakeholders (e.g. expected results, required success criteria).

7.3 Build a realistic schedule

Although closely related to the scope and target of the test effort, the schedule is worth discussing separately. Long-haul tests take time to design and execute and the schedule must reflect this explicitly. If the team agrees that, say, the Beta 2 product release must undergo 24 hours of long-haul testing, the team must also be aware that the Beta 2 release may be pushed out by several days due to last-minute issues found by the tests. Failure to account for these “test resets” can introduce unnecessary risks and uncertainty into the product cycle.

7.4 Iterations: crawl, walk, run

Long-haul tests themselves are iterative, always inching closer to an end goal through a series of smaller steps. The long-haul initiative should be no different. Consider defining a set of “crawl, walk, run” goals to slowly build up the capabilities, breadth, and rigor of the tests. It is important to focus on small, realistic objectives, especially if the organization has never done long-haul testing before. The full progression to the final goal may take days, weeks, or months; the key is to ensure the timelines and associated goals are well-defined, well-known, and agreed upon within the organization.

For example, the very first long-haul tests may set a target duration of only four hours, validate only that the product does not crash, and cover only one major feature area. After allowing sufficient time for the product and tests to stabilize, the target can increase steadily toward a final goal of, say, 24 to 48 hours of continuous operation with a full set of validations covering multiple features and subsystems.

7.5 Hold the bar

Long-haul testing can be quite challenging at times. Tests may find hard-to-diagnose bugs at inopportune moments. Resist the temptation to relax the exit criteria or ignore issues. If the team has set the bar and decided on specific long-haul goals, do all that you can to **hold the bar**. Make reasonable exceptions if necessary but always involve the project stakeholders and be transparent about it. Sometimes a late-breaking issue is truly an outlier and should not block the release; always assess the risk and make an informed decision as a team.

Similarly, do not subject the team to a moving target. Avoid making major or frequent changes to the scope or validations of the long-haul tests without first engaging with those who will be impacted (e.g. developers, release managers). Long-haul testing can pay dividends for almost any team but it will have the best impact if the effort evolves in an intentional, incremental way throughout the project lifecycle.

References

[1] Beck, Kent and Erich Gamma. 2002. “Test Infected: Programmers Love Writing Tests.” SourceForge. http://junit.sourceforge.net/doc/testinfected/testing.htm (accessed June 1, 2013).

[2] Analysis.Net Research. 2012. “7th Annual State of Agile Development Survey.” VersionOne. http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf (accessed June 1, 2013).

[3] Page, Alan and others. 2008. *How We Test Software at Microsoft*. Microsoft Press.

[4] Whittaker, James A. and others. 2012. *How Google Tests Software*. Addison-Wesley Professional.

[5] Osterman Research, Inc. 2007. “Planning for Improved Email Availability.” Osterman Research, Inc. http://www.ostermanresearch.com/whitepapers/or_nev0707.pdf (accessed June 2, 2013).

[6] U.S. Food and Drug Administration. 2009. “Glossary of Computer Systems Software Development.” U.S. Food and Drug Administration. http://www.fda.gov/ICECI/Inspections/InspectionGuides/ucm074875.htm (accessed June 2, 2013).

[7] Cherskov, Sergio. 2003. “Windows CE .NET Long-Haul Testing Scenarios and Results.” Microsoft. http://msdn.microsoft.com/en-us/library/ms836785.aspx (accessed June 2, 2013).

[8] Microsoft Support. 2012. “Description of race conditions and deadlocks.” Microsoft. http://support.microsoft.com/kb/317723 (accessed June 8, 2013).

[9] Beatty, Sean M. 2003. “Where testing fails.” Embedded.com. http://www.embedded.com/design/embedded/4024600/Where-testing-fails (accessed June 8, 2013).

[10] Mahmoud, Qusay H. “Using Assertions in Java Technology.” Oracle. http://www.oracle.com/us/technologies/java/assertions-139853.html (accessed June 8, 2013).

[11] Shore, Jim. 2004. “Fail Fast.” Martin Fowler. http://www.martinfowler.com/ieeeSoftware/failFast.pdf (accessed June 5, 2013).

[12] Kaner, Cem. 2004. “Examples of Test Oracles.” Center for Software Testing Education & Research. http://www.testingeducation.org/k04/OracleExamples.htm (accessed June 8, 2013).

[13] Hoffman, Douglas. 1998. “A Taxonomy for Test Oracles.” Software Quality Methods, LLC. http://www.softwarequalitymethods.com/Papers/OracleTax.pdf (accessed June 8, 2013).