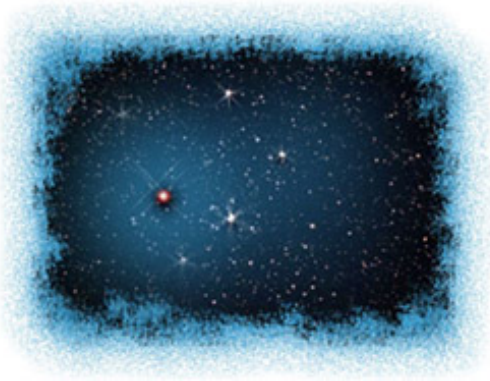
# The Revolution in Software Testing

by **Sam Guckenheimer**
Senior Director, Technology
Automated Software Quality
Group
Rational Software



*When Albert Einstein published the General Theory of Relativity in 1915, it was a grand work of conjecture. Four years later, Arthur Eddington and a team of British scientists conducted an experiment in which they photographed the star cluster Hyades during a solar eclipse. The experiment seemed to confirm (subject to a large margin of error) Einstein's predictions about the curvature of space and the effect of gravity on light. The popular press made Einstein and Eddington instant celebrities, perhaps because, as pacifists, they were both well cast as heroes for a war-ravaged world.*

*Although the press did not wait, it's worth noting that the General Theory of Relativity was still quite controversial in the scientific world at that time. Conclusive experimental results did not come until a half-century later, shortly after Thomas Kuhn wrote* The Structure of Scientific Revolutions. *Still, relativity is the perfect poster child for a paradigm shift -- what Kuhn called the discontinuity that occurs when one set of beliefs is overturned and replaced by a new one.*

*In July, when I interviewed Cem Kaner for* The Rational Edge, *he borrowed Kuhn's framework to categorize the conflicting, pre-scientific paradigms that prevail in the world of software testing today. Subsequently,* The Edge *published several more interviews that I held with experts on other aspects of software testing. Some readers have questioned my selection of topics. "What does this have to do with the mainstream work I do?" they've asked. So in this issue, I'd like to pull all of these topics together and explain my personal vision of testing in the future.*

I predict that testers, developers, project managers, CxOs, and end-users are all going to see big changes in software test practice this decade. It's

not hard to figure out why -- poor software quality currently costs the US economy anywhere from $60 billion (the NIST estimate[1]) to $200 billion (the Standish Group estimate) annually. So improving software quality has become a straight ROI play: Those enterprises that master software quality will win; the rest will be forgotten.

What will the practices and tools look like? I think they will be based on five current trends that will work together over time.

1. Test-driven development at all levels of the software lifecycle, leading to the convergence of test, requirements analysis, and specification with visual notation, based on UML and emerging standards.
2. Exploratory learning and discovery as a respected part of the iterative development process.
3. Component testing and design for testability as inseparable parts of development.
4. Attention to appropriate skills rather than prescriptive recipes as the basis for good process.
5. Automation of enormous layers of grunt work that impede effective testing today.

Let me describe these converging trends.

## Test-Driven Development

This practice is also called "test-first design" (in the Rational Unified Process,® for example) or "test-first programming" (in many eXtreme Programming articles). It has been with us for decades, but recently gained new momentum at the developer level, thanks largely to the Agile community. Their core idea is that, before you write a line of code, you must have written a test that fails. The test contains one instance of an expected behavior for the code to be written. Martin Fowler has usefully labeled such a test a "specification by example."

Brain Marick and other proponents of Agile Testing have proposed extending the idea of test-driven development to cover all levels of abstraction, including system/product-level tests.[2] Marick describes his goals pretty clearly: "I do not try to write a set of requirements that captures [the customer's] desires. Instead, I write a set of tests that, once they pass, will satisfy her desires. So I'm leaving out the requirements writing step and folding requirements analysis into the creation of tests."[3] The test acts as an executable specification; when the code passes the test, the code conforms to specification.

If your code is in Java and your tests are in Java, perhaps based on JUnit, and you're either the one person or half of the one pair of people writing both, it's easy to see that Marick's approach would work. Marick believes that this can scale to small workgroups with an onsite customer through the practice of "conversational test creation." However, when someone

else needs to understand the requirements (which are captured in the tests), and you're not there to explain them, you run into an obvious communication problem. By the way, I don't think the primary issue is that the tests are represented in a programming language; there would still be an understandability problem if you tried to capture them in "precise" English. This problem is well described by Leffingwell and Widrig;[4] Figure 1 is based on their ideas.

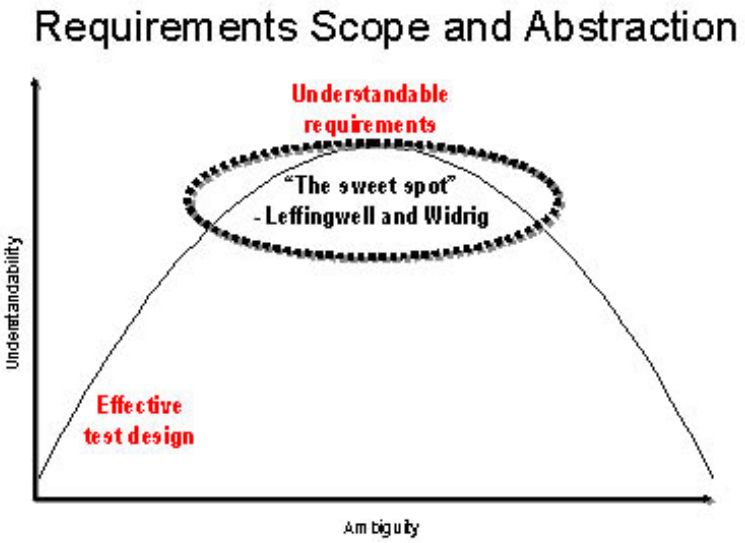



**Figure 1: The Understandability Problem (Based on Leffingwell and Widrig)**

Actually, Leffingwell and Widrig don't really consider the test issue; they just want requirements to be sufficiently high-level to be understandable to customers/stakeholders, and they leave the problem of adding specificity to other stakeholders and other parts of the lifecycle. Controversially, they write about leaving in some ambiguity. And the very idea of ambiguous requirements, of course, drives testers crazy.

The test-driven development approach that Marick proposes goes to the opposite extreme: Tests are the representation of requirements, in the form of compilable and executable code. Yet, if the only representation of your tests (requirements) is code, then you cannot communicate effectively with businesspeople/stakeholders/customers (or probably with other testers or programmers, either). All of these people think of tests in terms of data and flows, so your tests would be more communicable if they could be expressed that way.

Several companies are working to help bridge this gap. Rational is actively involved in an OMG working group on the UML Test Profile, which will represent tests as data and visual flows, such as sequence diagrams and activity diagrams. And we're producing tools that will treat the three representations (code, data, and flow) as alternate views of the same tests.

In the past, we drew these "specifications by example" and called them, according to RUP, "use-case realizations." The similarity between specifications by example and use-case realizations is made very clear in excerpt from an experience report from a recent workshop on Agile

Methods:

> [A participant] mentioned that the people he works with love test-first development. They do it easily once the test framework has been developed especially when test scripts have been defined to order the execution. **Use cases help when weaved together to develop the scripts. Tests by their nature capture user expectations. The reason people love it is because it gives them the structure they need to work under. With agile, requirements are in the form of stories. Therefore, test scripts weave the stories together** in a fashion that interfaces with others [sic] are explicitly defined. This means that pairs can work with others to test out the interfaces in the order in which they need to.[5]

Similarly, the OMG Test Profile working group has discovered that it is easy to align the UML representations as well. You can turn a use-case realization into a test by adding just two things: a validation action (e.g., "Now check this condition in the Software Under Test") and a verdict (e.g., pass, fail, or inconclusive). This information was captured in specification anyway, so UML notation lets us testers have our cake and eat it, too.

So now, with very little extra effort, we have tests that are understandable to customers: tables of data and visual flows. Later on, when there's software to test, the tests are executable at the same time. This practice makes it practical to have test-driven development at the system level, because there is really no distinction between the design artifact and the test artifact. Just add the annotations for validation and verdict, and you're there. With understandable visual flows and data, you no longer need to separate system design from test design. And because these flows have an executable form (the generated code), they can be run as a test against every build. This frees the whole team to become what Kent Beck and Erich Gamma call "test infected":

> ...a style of testing that with a remarkably small investment will make you a faster, more productive, more predictable, and less stressed developer.
>
> -Kent Beck and Erich Gamma
>
> From *Test Infected: Programmers Love Writing Tests*[6]

## Exploratory Learning

This second trend recognizes the reality that we don't always specify correctly: that requirements evolve, and we need to either flatten or invert the famous cost-of-a-defect curve. The central idea here is that everything I said about the visual-and-executable-design-and-test process should also be true of discoveries you make while running the software, whether in production, test, or the first step-through in the debugger.

Runtime Analysis tools, such as Rational® PurifyPlus, have a well-established following for very specific instances, such as finding memory errors and performance bottlenecks. But you can also use them to support a broader style of exploration, particularly as systems are assembled from

diverse components. And 80 percent of Enterprise IT activity is based on component assembly, including legacy renewal, rather than green-field design.

You can discover at least as much information from software when it is running as you can while designing *de novo*. (That's why RUP emphasizes executable software as part of every iteration.) And everything you discover should be viewable in the same artifacts you used for design. The trace of the running system is a UML interaction, and it may be worth turning into a reusable test by adding that validation and verdict. The data may be worth generalizing into the seed of a new equivalence class. And so on.

Currently, most advocates of exploratory testing, notably Kaner and Bach, treat it as a throwaway activity,[7] but I think we'll discover that it's highly reusable, once we can move the learning seamlessly into those design artifacts. In fact, this is a new use for Runtime Analysis: the capture of assets discovered through exploration.

## Component Testing and Design for Testability

The third trend is about understanding the relative roles of testers and developers and providing the right tooling for each of them. Rational has long promoted component testing and design for testability as best practices, and I think adopting them stems from a basic understanding of quality. Today, too much of testers' activity is stuck in conformance-to-spec mode, which is a waste. *Developers* should ensure that the software conforms to spec -- and they should have the tools and process to make that painless.

Boris Beizer expressed the difference between the two roles as follows:

> *The purpose of independent testing is to provide a different perspective and, therefore, different tests; furthermore to conduct those tests in a richer...environment than is possible for the developer. The purpose of self-testing is to eliminate those bugs that can be found at a lower cost in the simpler, deterministic, environment of the unit/component or low-level system test.*[8]

Test-driven development is a great enabler. If the specs are executable tests, and the tests run green, then the software conforms to the spec. Other unit testing processes are meant to ensure the same thing: As far as we can tell from the specifications (whatever they are), the code conforms. And if it doesn't conform, it should be treated as a development problem. Kent Beck is pretty clear about this effect of test-driven development:

> *If the defect density of test-driven code is low enough, then the role of professional testing will inevitably change from "adult supervision" to something more closely resembling an amplifier for the communication between those who generally have a feeling for what the system should do and those who will make*

*it do.*[9]

When a team accepts the premise that conformance to spec is a developer responsibility, it frees the testers to look for anything else that might diminish the perceived value of the software from the customer's or user's perspective. In other words, what things did no one think of in the specification? Brian Marick wrote a great paper that discusses these errors of omission.[10] They usually account for more than half of the errors in a system, he claims, so you'd better have a process that enables your testers to look for them.

Design for testability can play an important role here. Software everywhere is moving to services-based architectures, which are an extension of component-based architectures, with the additional complexity that the components can change without warning. The reliability issues are enormous. Most IT managers would accept 99 percent reliability, and I bet most would say that's a higher standard than they apply to components they buy and build. Yet if you architect a system out of 100 components, each of which has 99 percent reliability, then the reliability of the whole is $(0.99)^{100}$, or about 37 percent. (By the way, that's why high-reliability markets like telecom look for "five nines" of reliability, or 99.999 percent. That way, you can use 100 components and still achieve 99.9 percent composite reliability.)

This fundamental principle begs the need for design for testability in software, just as it did in hardware, when a component marketplace emerged there thirty years ago. Bertrand Meyer took the lead in this area when he advanced the concept of *Design by Contract*, which means viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations.[11] A sign that Meyer's concept has gained acceptance is the presence of a design-by-contract specification language, WSDL, at the core of the Web Services standards[12].

I think that people are embracing design-for-testability more and more. In addition to becoming part of standards and frameworks like Web Services, interfaces are becoming part of technology platforms and operating systems. A simple example is the opening of profiling interfaces and reflection APIs in J2EE and .NET that allow tools to inspect what is happening in the runtime environment. Another very real and beneficial trend is that people are building applications from design patterns now with tools like Rational® XDE™ -- and testability is built into the design pattern. Components constructed from patterns include exposure to test interfaces -- with appropriate getters and setters on the components.

One of the practical principles of design-for-testability is that you have access to the business logic or behavior of the software under test, just beneath the GUI, or presentation layer. Bret Pettichord contends that design-for-testability is about visibility and control.[13] You get the visibility you need through exposure at lower layers, with lots of open interfaces that allow you to see into the state of the software under test. Similarly, you need interfaces that allow you to control the application so you can drive it from an automation framework without engaging the GUI.

## Attention to Skills

The fourth trend is the increasing level of knowledge and skill expected of software testing professionals. It was a common misconception during the dot-com boom that you could test effectively without having deep technical knowledge of the software under test, deep domain knowledge about the business application, or much training at all. But when you look at a distributed application -- a Web application in particular -- that assumption breaks down. Hung Nguyen's book on testing Web-based applications was a landmark with respect to this point.[14] Testers should know how technology affects the kinds of errors and risks they can see, Nguyen explains. They need an understanding of both technology issues -- such as the deployment topology -- and the kinds of errors inherent in the technologies they're examining. Even understanding details such as the difference between bean-managed and container-managed persistence on an application server, for example, can affect your ability to detect certain kinds of faults.

So today's testers need to understand the technology and the domain as well as generic testing techniques. For example, suppose you see an error message in the browser that says "404 - Page not found." That error might be caused by a broken link, or it might occur because some service has become unavailable. A good tester will not stop with the error page; he'll go on to diagnose the cause. Not only will he know enough to suspect the unavailable service, but also he'll be able to confirm his suspicion -- by looking at other pages that depend on that service, for example. This is a critical technique for isolating a bug.

Another timely skill is the ability to be a good explorer. Historically, a lot of what has been described as testing was very scripted and planned, but in reality good testers are good explorers. They recognize hints and know how to follow up on them. The hint might be something as simple as a page that takes surprisingly long to load. A good tester will ask, what's going on here? And then know what paths to go down to find out. James Bach has written the best material about exploratory testing[15] and has the best exercises on the subject. I think it certainly is a critical skill, and one that every testing team should have.

We've focused heavily on building basic software testing skills in our Rational University curriculum, working with Cem Kaner of Florida Tech to develop a new course on the Principles of Software Testing for Testers.[16] The course focuses not on tools, but rather on being a better software tester, especially if you use an iterative process. Ultimately, the differences in productivity among testers are at least as great as those among developers, and it's the skilled testers who will help produce a great ROI for their customers.

Since its founding, Rational has believed that the key to faster, more cost-effective, and higher-quality development is an iterative development process that brings testing forward in the development cycle, making it possible to find defects when they are cheaper and easier to repair. Right now, however, I don't think testers are well trained to work within an iterative process. Nor are project managers well trained to consider the

testing role; and developers are not well trained in the testing techniques they need to know, such as basic equivalence partitioning. That's why we've added a lot of material on testing to the Rational Unified Process and the Rational University curriculum; and we'll continue to amplify this material as we help train testers, developers, and project managers to work together, iteratively.

## Automation

The fifth trend is about automation. Today, testers and developers expend 80 percent of their effort on make testing possible, and only 20 percent on making testing *meaningful*. (This formidable overhead leads many to dismiss the value of automated testing altogether.) Similarly, today's automated software quality (ASQ) tool vendors are expending 80 percent of their effort on duplicative work, recreating an infrastructure to enable testing and debugging activities; only 20 percent produces functionality that is visible and valuable to testers and developers.

Recently, Rational, IBM, and a few other companies began work on an open-source tools project that aims to reverse these percentages. This Hyades project, named after the star cluster Eddington used to validate Einstein's theory, will be announced this month, and is organized as an initiative of Eclipse.org. It also aims to enable empirical observation, testing, and measurement of software -- to make good test automation just as practical as it is theoretical.

For developers and testers using Eclipse, Hyades is an integrated test, trace, and monitoring environment that will provide standards, tools, and tool interoperability across the test process, thereby moving testing earlier into the application lifecycle. For ASQ tool vendors and integrators, Hyades will deliver an extensible framework and infrastructure for automated testing, trace, profiling, monitoring, and asset management. Unlike the test and trace tools available today, Hyades will provide a unified data model (implementing the UML Test Profile), a standard user experience and workflow, and a unified set of APIs and reference tools that work consistently across the range of targets.

## Overall: A Great Change in Practice

Why would Rational and several of its competitors, all of whom produce commercial testing tools, embrace and contribute to an open-source project like Hyades? (Believe me, many of my colleagues have asked this question, too.) The core reason is that 80 percent/ 20 percent ratio I just described; everyone wants to change this.

The 80 percent infrastructure is invisible to users; it is not differentiated, and it is a nuisance to maintain. The tools need to be updated every time the technology of the software under test is updated (with new compilers, new libraries, new OS service packs, etc.) If you're an experienced user of test automation or runtime analysis tools, you've probably experienced this fragility. Chances are, you've hesitated at least once to upgrade your development environment because some tool didn't support the new release.

This maintenance cost creates an enormous drag coefficient for vendors. By sharing a common chassis, we are freed to work on the engines and dashboards that matter to users. Hyades should accelerate the delivery of value for our users.

By itself, Hyades is a discrete effort. In the context of the five different trends I've outlined, however, Hyades is part of a paradigm shift to a new breed of technology that supports a new sort of testing for both testers and developers. It's a technology that promotes testing at the beginning of the lifecycle, more interoperability of tools, and more visibility into the software under test. It supports a greater change in practice than we have seen in decades. I believe this technology -- and others with similar goals and foundations -- represents the future of our industry. Those of us involved with the Hyades project are pushing for it to live up to the reputation of its namesake:

> Outlining the head of Taurus the Bull, stars in the Hyades cluster are important to us. Oh, they give us pleasure to behold, but they also enable us to measure the universe.
>
> --Anthony G. A. Brown, Universidad Nacional Autýnoma de Mýxico.

---

## Notes

[1] http://www.nist.gov/director/prog-ofc/report02-3.pdf

[2] Kent Beck draws a distinction, calling Marick's idea "Application-Test-Driven Development". Kent Beck, *Test-Driven Development*. Addison- Wesley, 2002, p. 199.

[3] http://www.therationaledge.com/content/oct_02/f_testFirstDesign_sg.jsp

[4] Dean Leffingwell and Don Widrig, *Managing Software Requirements*. Addison-Wesley, 2000, p. 273.

[5] http://fc-md.umd.edu/projects/Agile/3rd-eWorkshop/topic4.html

[6] http://junit.sourceforge.net/doc/testinfected/testing.htm

[7] Their course materials are now publicly available at http://www.testingeducation.org/ Bach's Web site www.satisfice.com is also a good resource.

[8] Boris Beizer, *Black-Box Testing*. Wiley, 1995, p.13.

[9] Beck, *Test-Driven Development*. Addison Wesley, 2002, p. 86.

[10] http://www.testing.com/writings/omissions.pdf

[11] Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997, p. 331.

[12] See http://www.w3.org/2002/ws/ and http://www.ws-i.org/

[13] See, for example, the discussion in

http://www.therationaledge.com/content/nov_02/f_pettichordInterview_sg.jsp

[14] Hung Q. Nguyen, *Testing Applications on the Web*. Wiley, 2001.

[15] http://www.satisfice.com/articles/what_is_et.htm is a good starting point.

[16] For a good discussion of this work, see
http://www.therationaledge.com/content/jul_02/f_interviewWithKaner_sg.jsp

---

***For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!***