# Going Faster: Testing The Web Application

**Edward Hieatt and Robert Mee,** *Evant*

This article documents test-first design and the creation of testable code for Web applications. The authors explain how testing has been critical to building Evant's application at speed while maintaining a high degree of quality.

Testing is a fundamental aspect of software engineering, but it is a practice that too often falls by the wayside in today's fast-paced Web application development culture. Often, valuable software engineering principles are discarded, simply because they are perceived as being too time-consuming and lacking a significant payoff, and testing is a common casualty. Testing is often last in developers' minds when pressured to deliver something, anything, before the competition jumps on it, the market changes, or the money runs out. Despite developers' hard work, the resulting code is often precisely the opposite of what is desired: unstable and fragile, profoundly slowing development. Furthermore, adapting to new requirements is difficult because existing functionality must be manually retested whenever a change is made. We know that testing is the answer, but is there a way to adhere to doing it while still developing at a fast pace?

We have found that a radical approach *centered* on testing proves highly effective in achieving rapid development. We have spent the last two years using the Extreme Programming methodology to develop e-commerce software at Evant, a Web-based ASP company. Focusing on testing and testability results in a code base that can be built on quickly and that is malleable to the extent that it can easily be changed to accommodate new customer requirements and the latest Internet technologies. We describe how testing helped and how we established it as the development centerpiece.

## Testing first

First, let us briefly describe what we mean by "test-first" programming. To test-first means to write a unit test for a new piece of functionality before the new code is written. For example, if one were writing a "shoot-'em-up" computer game, one might write a test asserting that the player initially has three lives before the code is implemented to make that the case.

Using unit testing to drive development rather than relying solely on a quality assurance team profoundly changes several aspects of software engineering. Designs often differ from what they might have otherwise been, technology choices are influenced, and the way in which the application code is integrated with third-party frameworks is significantly altered.


0740-7459/02/$17.00 © 2002 IEEE

Though it might seem a tautology, practicing test-first programming leads to a pervasive quality of testability throughout the code base. By having tests in place, the overall code design is positively affected. More tests, and more features, can easily be added to designs and beneficial characteristics arise as a natural consequence of its testability.

One such beneficial characteristic is reusable code. Code that is developed for a single purpose usually services a single client (meaning client code); a test provides a second client for the interface. Thus we force reuse by testing. This secondary interface exercise, and its resultant refactoring, tends to flush out design flaws that might exist in single-use code. For instance, a class B written to support class A might make inappropriate assumptions. Writing unit tests specifically for B, independent of its relationship with A, removes these assumptions, which leaves classes A and B more loosely coupled and therefore more reusable.

Testable code tends to have a cleaner interface, and classes tend to be defined at a more appropriate granularity than they might otherwise be. Writing tests simply makes it more apparent when a single class's scope is too ambitious or too reliant on the workings of another.

### Testing the servlet

Unit testing has beneficial effects on removing dependence on a particular technology. Often software teams are directed to minimize their reliance on a single tool, technology, or vendor in an effort to maintain long-term flexibility. This is easier said than done. Take the ubiquitous Java servlet as an example. The instructions are simple: override the "`service()`" method and have it do what you want. Several outcomes are possible as the complexity of request processing grows or the types of different requests increase. The "`service()`" method might become long or break up into several methods in the class, there might be many utility methods added to the servlet subclass, or a whole hierarchy of different subclasses might arise in an attempt to specialize and still reuse code. The problem is that servlets are extremely awkward to test, primarily because they are used as a component in a specific environment—the servlet container. Creating instances of servlets and testing individual methods is quite difficult.

The test-first approach leads to a different implementation. When it comes to servlets, it is quite apparent that the easiest way to test the code that handles requests is to remove it from the servlet entirely, and put it into its own class. In fact, not only is that the easiest thing to do with respect to the tests, but the resulting design itself is then more flexible and quite independent of the servlet technology. Our code in "`service()`" is three lines long. It has just one responsibility: create another object that will do the work and then hand control to it. In our case, we call this object a *dispatcher*. Its job is to decide what kind of request is being submitted and create yet another object (an instance of a command pattern) to handle the request. The dispatcher and each command object have their own unit tests, in which creating the needed fixture is trivial. Now we not only have a testable design, we have one with a more appropriate division of labor and one that is more technology-independent.

### Aggressive refactoring: Swapping technologies

With the support of extensive unit tests it is possible to radically change code without breaking existing functionality. This gives the developer the ability to make major changes, even in integral frameworks, very quickly. For example, it allowed us to refactor our persistence code just weeks before we went live with our first client.

We used Enterprise JavaBeans, specifically Entity beans, to manage our database persistence. We wanted to remove the entire EJB layer because of poor performance and difficulties in testing the internals of the Beans, among other reasons. Without tests, we would not have attempted to remove the Beans, but given that our unit test suite included extensive coverage for testing persistence, and that we had experience in performing this kind of major change, none of us was too concerned. In fact, we would be getting a performance improvement, more testable code, and the removal of an expensive technology.

We executed the refactoring in less than a week, and performance did increase markedly. We saved money by removing the EJB technology and ended up with more testable code because we had removed a part of our system that was essentially a black box.

### Bugs

Testing has radically changed the way we

**Though it might seem a tautology, practicing test-first programming leads to a pervasive quality of testability throughout the code base.**

**The existing tests might not be extensive enough, in which case the programmers would add more tests.**

deal with controlling bugs. We think of bugs differently from the traditional view—we use our bug count to help pinpoint areas that need more testing. That is, we consider bugs as feedback about how we are doing with our testing.

When a bug is found, we write a test that "catches" the bug—a test in the area where the bug was found that asserts what should be the case and fails. We then change the code until the test runs successfully, thus fixing the bug. In this way, we fill in holes in our suite of tests.

We do not stop there, though. We look for the case where several bugs in a related area entered our bug tracking system during a short time period. This information tells us that something more seriously wrong exists with our tests—perhaps the testing framework in that area needs some work. For example, perhaps three bugs are found one day all in the area of logging out of the application. Rather than trying to "pinpoint fix" each bug—that is, assign each bug to a different developer, thinking of each as an independent problem—we assign the group of bugs to a single pair of programmers. This pair looks at existing tests for that area, which clearly did not do their job well. The existing tests might not be extensive enough, in which case the programmers would add more tests. Or they might refactor the testing framework in that area to allow better, more accurate support for the kinds of tests that we can then proceed to write.

The result is a suite of tests that constantly grows and changes to cover as much of the application as possible. The tests are run whenever code is changed, catching problems before the changes are even deemed worthy of being added to the main code base. Armed with such a weapon, as little time as possible is spent maintaining existing functionality. In short, development occurs faster.

### Tests as documentation

Our methodology does not encourage writing documentation for code or even writing comments in the code. Instead, we rely on tests to document the system. This might seem strange, but because tests are written first, they completely define what the code should do. (In fact, the definition of the code being in working order is that the tests all run.) We write our tests with readability in mind. The best way to learn what the code is supposed to do is to read the tests. New developers, or developers who are inexperienced in a particular area, can get up to speed far quicker this way than by trying to wade through requirement and design documents. It is more effective to learn by reading a short, simple test than to try and decipher code that might contain problems or is very complex.

This method of learning through reading tests is especially important in our company as the development team grows. As new developers come on, they are paired with existing developers to gain knowledge, but a large part of their learning comes from reading tests. New team members come up to speed quickly because of well-factored, readable tests.

### Acceptance tests and their extensions

In addition to writing unit tests, our process has a mechanism called *acceptance tests* that allow product managers to express tests at the scenario level.

We began by writing a testing tool that expressed operations in the system and the expected results as XML. Product managers, with the assistance of programmers and quality assurance team members, wrote extensive acceptance tests that, along with our suite of unit tests, are run at every integration.

The acceptance tests provided a clear value, but it was soon apparent that the supporting frameworks for the XML had several other benefits. As we neared deployment for our first customer, the need arose to integrate with other e-commerce systems and with the client's legacy mainframes. We employed an industry-standard EAI (Enterprise Application Integration) tool configured to generate XML as an adaptor to transform data from these external systems to our own. The XML test framework required little modification to act as the interface to this integration tool. We even used the framework assertion capabilities to verify incoming transactions.

As our need for acceptance tests grew, writing them more quickly and making them easily maintainable became important. XML, though workable, is awkward for a human to write, especially for nontechnical staff. To solve this, we implemented a domain-specific language, called Evant Script Programming (ESP), which is more compact and readable than XML. Once ESP was available, we recognized uses beyond acceptance tests and

data loading, including an easy way to configure system parameters, and that we could use a suite of ESP scripts to create a tailored demo environment.

## Testing challenges and solutions

In the Web environment, the client and the server are very different beasts. The user's machine runs a Web browser, which understands HTML and JavaScript (we will not discuss the extra complications of applets, plug-ins, or other client-side technologies here). The only ability this browser has is to send information through a map of strings over a stateless protocol to a Web server, which deals with the submission by passing it off to an engine running application code (for example a servlet container running Java code or a Perl program). We would typically like to test the client and the application server part of this set up, but it would be difficult to come up with a testing framework that could do both at once—to test JavaScript and DHTML on the client and Java on the application server. However, splitting tests into two distinct parts is not a good idea because we need to test how the two sides interact.

This is the classic Web application testing problem. We recommend the following remedy.

First, test those parts of the server-side code that are not directly concerned with being part of a Web application, without involving the Web peculiarities. For example, if one were testing the manipulation of a tree object, one would test manipulating the object directly, ignoring the fact that the visual representation of the tree is collapsed and expanded by the user through the UI. Test the business logic with such low-granularity unit tests. That is, of course, the goal no matter what the UI, but it becomes particularly important to be strict about it when working with a Web application.

Second, test those parts of the client-side code that have no server interaction. This is typically code that contains little or no business logic. Examples of such code, typically written in JavaScript, might include functions to pop up error messages, perform simple field validations, and so on. The immediate problem we faced in writing tests for this sort of code was that no testing framework existed for JavaScript inside the browser. In fact, we had already heard from other developers that testing JavaScript was too hard, and that JavaScript itself should therefore be avoided entirely. Undeterred, we wrote a testing framework for JavaScript called JsUnit, now a member of the XUnit family of unit testing frameworks (www.jsunit.net). JavaScript is notoriously difficult to write and debug, but using JsUnit has greatly helped our rapid development of client-side code.

Third, write functional tests of a low grain in the server-side language (for example, Java or C++) that simulate the request/response Web environment. We wrote a framework for such tests that extends our usual unit-testing framework to allow the test author to think of himself or herself as being in the Web browser position, putting keys and values into a Map object (which corresponds to `<form>` in an HTML document). The framework simulates a submission of the Map to the server by invoking the server code, passing in the Map as a parameter. This invocation is coded in such a way as to follow as closely as possible the path a request would take were it submitted through the true UI. (We take advantage here of the refactoring of our servlet discussed above: We cannot "invoke" the servlet from a test, but we can easily talk to the objects that the servlet spins off.) The server's response can then be inspected and verified by the test (see the next section for more on this). Because the test code explicitly invokes the server-side code, the test can inspect the state of the code, rather than just the HTML output resulting from the submission. It is for this reason especially that this method of testing is more useful than a framework such as HTTPUnit, which views the situation purely from the client's viewpoint.

## For More Information

**Books**

*Extreme Programming Explained,* Kent Beck, especially Chapter 18 ("The introduction to Extreme Programming"); a good place to start.

*Extreme Programming Installed,* Ron Jeffries et al., Chapters 13, 14, 29, and 34; contains a lot of information addressing common problems and questions about unit testing.

**Web sites**

Testing frameworks:
www.xprogramming.com/software.htm
www.junit.org

Extreme Programming sites:
www.xprogramming.com
www.extremeprogramming.org

**How to test output from the server is another classic problem because of the nature of Web applications.**

Using this framework, you can write walkthrough tests that simulate the user moving through the UI. For example, we have methods available in the test framework with names such as "`clickSaveButton()`." A test case might call that method and then perform assertions about the domain object that should have been persisted as a result of the call. Thus, we have the ability to perform an action that the user might execute and then to test specifics about the domain. Walkthrough tests are particularly useful when writing tests to reproduce bugs that manual testing has exposed. In fact, there has not yet been a case when we could not write a walkthrough test that catches a bug found by a member of the quality assurance team.

### Testing server output

How to test output from the server is another classic problem because of the nature of Web applications. How do you test HTML? Certainly we don't want tests that assert that the output is some long string of HTML. The test would be almost unreadable. Imagine tracking down a mistake in such a test, or changing it when new functionality is required. Testing the Web page look is not usually the goal; testing the data in the output is. Any slight change to the output—for example, a cosmetic change such as making a word appear in red—should not break an output test. We came to understand these issues as we progressed through several technologies used to generate our HTML output. In fact, this progression and the choice on which we finally settled were influenced in large part by what was most easily testable.

At the outset we worked with JavaServer Pages (JSPs). We found that although we got up and running quickly using them, testing them was hard. They are another Web technology that runs in a container environment that is difficult to reproduce in a test. In addition, it was challenging for our UI designers to work with JSPs because of the amount of Java code interspersed among the HTML. Soon we were spinning off testable objects to which the JSPs delegated that did the HTML generation. When we arrived at the point where the JSPs were literally one line long, we dispensed with them altogether and produced our HTML purely using Java. With that arrangement we could easily test the state of objects that produced the HTML—for example, that the object knew to make a button disabled. We could also test the process of producing pieces of HTML—for example, the production of an HTML `<table>` element. However, we were still not testing the output. The same question dogged us: How do you test the data in the HTML without getting confused with the cosmetics? Furthermore, our designers were even less interested in delving into Java code to change text color than they had been with JSPs.

A further change of technology solved both these problems. We generated our UI by outputting XML—pure data—from our domain objects. Then we transformed the XML into HTML for the client using Extensible Stylesheet Language Transformation (XSLT) pages, written and maintained by our developers and UI designers.

XML is easily testable. In our walkthrough tests, we can ask for the current XML output at any time and then make assertions about its structure and contents using the standard XPath syntax. The XML consists strictly of data from the domain; there is no knowledge even that it will be used in the UI. Our XSLT contains only information about how the UI should look and how to use the data in the XML.

Thus we achieved an extremely clean separation of testable business data from HTML generation that helped us develop the UI much faster than the other methods we tried. We could test that part of the output that contained data without getting it confused with the HTML. Conversely, our UI designers were happier because their work was not confused by Java among the HTML. Again we found a way to separate the untestable Web technology, in this case HTML, from the data.

## A first test

A question we are often asked is how we started promoting testing and testability at the beginning of our development. The first step in making our Web application testable was to establish a culture of testing among the developers. Following the methodology of writing tests first, our first task as a team was to write the first test. Most team members were new to the concept of test-first programming, so it was not obvious to them how to go about this.

It seems almost comical thinking back on it, but it was appropriate at the time: We pro-

grammed in a group for the first couple of days. Having the entire team of eight huddled around a monitor, achieving consensus on every line of code, let us iron out right from the start what a typical test looked like, how we write a little bit of test code and then some code to make the test run, when to go back and add another test case, and so on. We also quickly settled other issues that often plague development teams: we agreed on coding standards, established a way to perform integrations, and settled on a standard workstation configuration.

Thus testing has been an integral part of our development process since the first day, and remains so now. Of course, our testing frameworks and practices change and grow daily, and we are by no means finished with their evolution. For example, we feel that we have a long way to go in finding more appropriate ways to expose testing to our business users. We would also like to extend our walkthrough test framework to simulate even more closely what happens in real situations. Our company supports us when we ask for time to develop such abilities because they have seen the benefits of testing. They agree that our focus on tests and testability has helped us go faster than any project they have worked on in the past.



## About the Authors

**Edward Hieatt** is a lead developer at Evant, a software vendor in San Francisco that observes a strict practice of XP. He became involved with Extreme Programming in 1999. His research interests include the rapid development and testing of Web-enabled software. He is also the author of JsUnit (www.jsunit.net), a unit-testing framework for JavaScript, which is one of the XUnit testing frameworks. Contact him at edward@edwardh.com.

**Robert Mee** is a software programming and process consultant in a variety of industries, helping companies apply the practices of Extreme Programming. He regularly gives lectures on XP for corporations, their investors, and others. During 2000 and 2001, he was director of engineering at Evant, where he helped introduce XP. His research interests include human and computer languages, and the application of domain-specific languages to the automated testing of software. He has a BA in oriental languages from the University of California at Berkeley. Contact him at robmee@hotmail.com.