Increases speed: Doesn’t writing all these tests slow development down? Speed (or development cost) is one of the most frequent arguments against practicing TDD and using xUnit frameworks. All new tools have learning curves, but once your developers are comfortable with their frameworks of choice (often this takes a little time), development velocity will actually increase. A full unit test suite gives you a way to regression test your system, meaning you can sleep a little easier after adding that new whiz-bang feature.

Gives feedback: One often-overlooked benefit of unit testing relates to rhythm. It may seem trivial, but you gain a real sense of accomplishment when your tests pass! Rather than spending days and days hacking code without any feedback, the test-code-test approach encourages you to work in small chunks, often coding little more than a few minutes at a time. Instead of facing a daunting list of features, you inch your code base forward one tiny piece at a time.

In our experience, testing is infectious, and you can find yourself addicted to the green bar. Initially, many developers are skeptical, but inevitably the light bulb turns on for nearly every developer. Usually it’s when the tests catch a bug for the first time or when adding that new feature takes minutes instead of hours; regardless of when it occurs, usually developers realize that tests matter.

Introducing JUnit

Since JUnit serves as the inspiration for JsUnit, we’ll start with a brief overview of JUnit before diving into the details of JsUnit. For a truly detailed look at JUnit, take a look at any of the excellent books on the topic. While JUnit isn’t the only answer for testing (TestNG and Fit/FitNesse are worth investigating), JsUnit is essentially a “port” of JUnit for use in testing JavaScript, which is why we’re discussing JUnit here.

JUnit is one of the most widely used xUnit test frameworks. Written by Erich Gamma and Kent Beck, JUnit is commonly used on Java-based open-source software, and most common IDEs support it out of the box. Writing tests in JUnit is fairly trivial—you simply create a class that implements TestCase, write some methods that start with test and have some asserts in them, and then run your tests with your favorite test runner. By default, JUnit will automatically run any method that is prefixed with test, though you can override this behavior to suit your needs.

By the time you write your second or third test, you will find you have some common code that can be refactored. Since you’ve probably read The Pragmatic Programmer, by Andrew Hunt and David Thomas (Addison-Wesley, 1999), you know you shouldn’t repeat yourself, so you pull out the common code into a fixture by overriding the setUp() and tearDown() methods, which are called before and after (respectively) every test you run.

At first you may have only a few tests, but eventually their numbers will grow, and you’ll need some ways to organize them. In JUnit, you create TestSuites that consist of a collection of test methods or even entire test classes. (TestSuites can hold anything that implements the Test interface.) If you prefer greater control over your tests, you can manually add them to the TestSuite, or you can simply tell JUnit to do the work for you by passing your TestCase to the TestSuite constructor.

JUnit supports a number of test runners. Some IDEs have their own proprietary runners, and you are free to make your own if you want. JUnit comes with both a textual runner and a graphical runner that reports on the outcome of running your tests. (The graphical runner