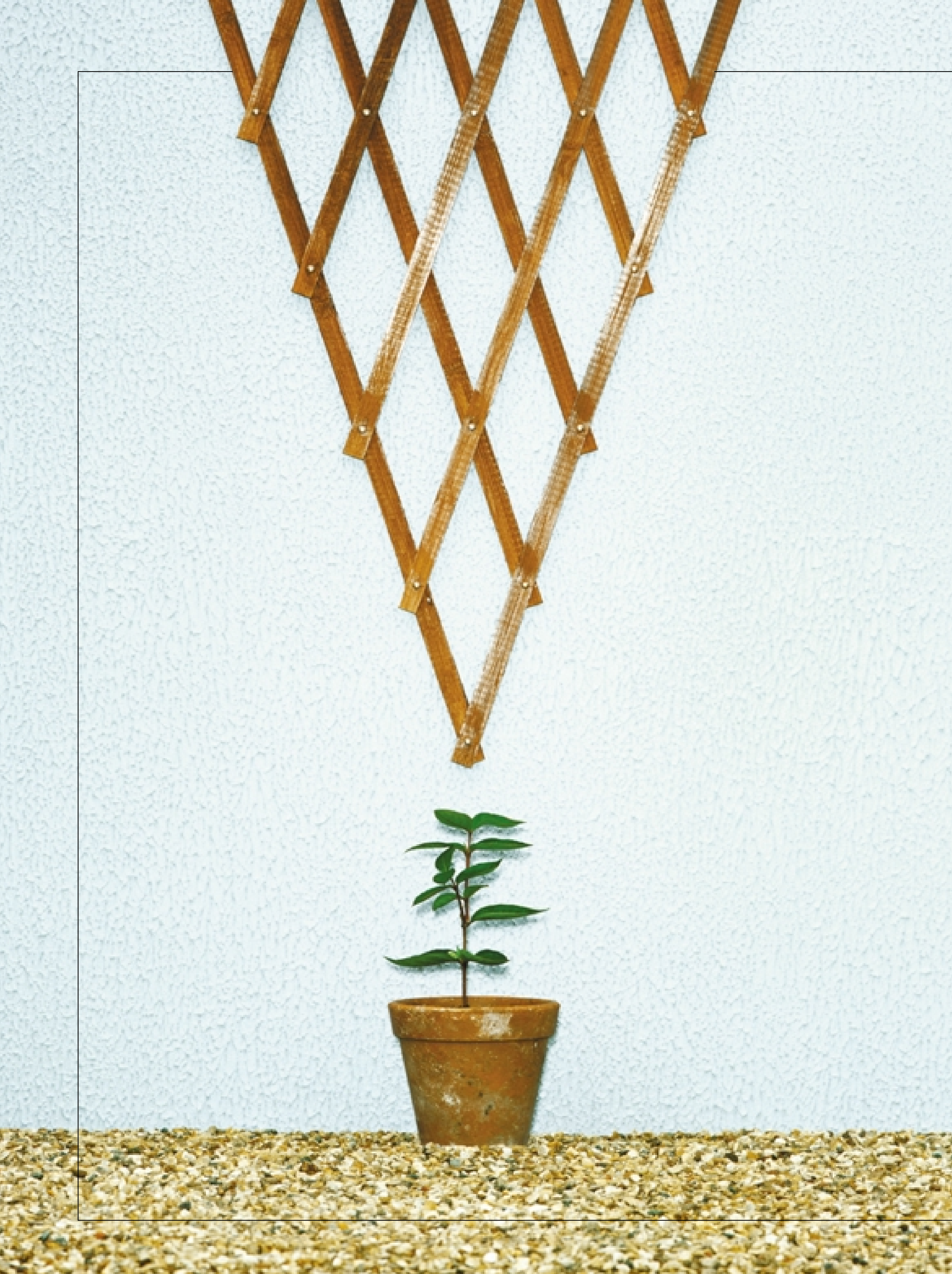

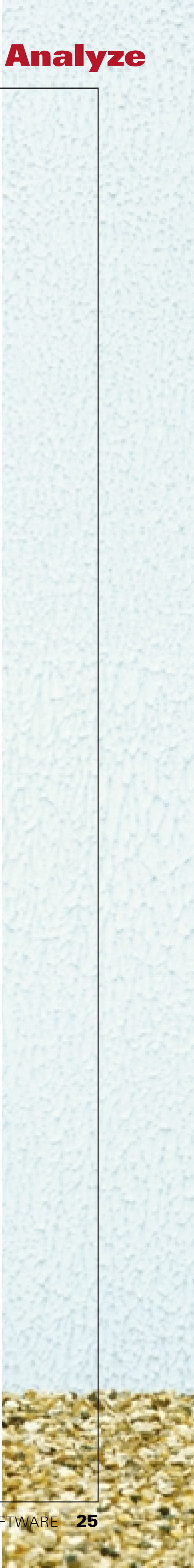

Building a test harness is an effort that often takes on a life of its own. But it doesn't have to get wildly out of control. Take a tip from Agile development and cultivate your harness, test by test, taking the time to

GROW YOUR TEST HARNESS NATURALLY

BY KEVIN LAWRENCE

I HAVE WORKED WITH MANY TESTING ORGANIZATIONS WHERE A common pattern is repeated over and over. It goes something like this—we realize there is more manual testing to do than time avail-

RICHARD DRURY/GETTY IMAGES

able, we decide to automate the testing, and we begin working on a test harness. Several weeks later, we have the start of a harness, but it's barely useful and we still have not written any tests. At this point, we're behind—so we abandon the harness and revert to manual testing.

The most common reason for abandoning a test harness is that the team writing it does not completely understand the needs of the team that will be using it. But even if there is a single team that will be both writing and using the test harness, the harness effort seems to take on a life of its own. Sometimes, the harness builders get so carried away with harness building that they never get around to test writing. It becomes a matter of pride that the harness should be able to deal with any possible testing requirement. No bell is too small; no whistle too trivial. Occasionally, the harness is actually completed but it sits on the shelf because it does not have the features that the testers need or because the testers don't have the technical skills to use it.

At my current company, we decided to try something different. We already enjoy the benefits of Agile planning and design and wanted to see if the same just-in-time philosophy would be as successful when building a test harness.

We had an extensive suite of unit-level tests but very few automated tests at the system or functional level. We wanted to start with a smoke test that could be run under our automated build system—CruiseControl—after every check in. (See Jeffrey Fredrick's article in the September 2004 issue of *Better Software* for more about CruiseControl.) A smoke test is a small set of system-level tests that run quickly and give you confidence that the product is ready for further testing. The name comes from hardware testing. If you turn the hardware on and smoke

Reusing the Test Harness

As important as the smoke tests are, the most valuable result of this project is a robust test harness that is actually useful. We use it extensively for functional testing and have found other applications for it, too. For example, whenever a bug gets submitted to our database, we create a test that demonstrates the problem. The harness makes the tests quick and easy to write and cuts out all of the "works on my machine" arguments.

One of the more novel uses of the harness was in automating what Hans Buwalda has called "soap opera" testing. (See Buwalda's article in the February 2004 issue of *Better Software*.) Functional tests can be more effective when they are designed with a particular scenario in mind, and soap opera testing takes this idea to the extreme. Like a soap opera on TV, a soap opera test exaggerates real-life scenarios and crams many of them into a single episode. Usually when you design tests, you try to isolate each test so it can run independently and not be affected by the tests that ran before it. A soap opera test can find those bugs that show up only after a series of seemingly unrelated operations.

We have a tutorial that takes new users on a whirlwind tour of Agitator, and it does so in the context of an application based on an order-entry system. I went through the whole tutorial manual and converted text like this . . .

To review the results for the constructor of the ***Product*** *class:*

1. *In the* ***Outcomes*** *view, double-click the constructor of* ***Product*** *to see its results. Agitator expands the node in the* ***Outcomes*** *view and lists the outcomes identified for the constructor. This method has a* ***NORMAL*** *outcome and two exception outcomes.*

2. *In the* ***Outcomes*** *view, select* ***NullPointerException****.*

3. *Click the* ***Snapshots*** *toolbar button in the* ***Outcomes*** *view to see snapshots for each occurrence of this outcome. On the* ***@EXCEPTION*** *line, the name of each exception thrown is a link to a stack trace that shows the state associated with the snapshots in that column.*

. . . into code like this:

```
void checkNullPointerSnapshots() {
    MethodResultHarness result = getMethodResult(CONSTRUCTOR);
    OutcomeHarness npe = result.getOutcome("java.lang.NullPointerException");

    SnapshotHarness snapshots = npe.getSnapshots();
    for (int i = 0; i < snapshots.getSnapshotCount(); i++) {
        SnapshotColumn snapshot = snapshots.getSnapshot(i);
        snapshot.assertStackTraceContains("Product.validateCode(Product.java:58)");
    }
}
```

. . . which is used in tests like this:

```
void reviewConstructor() {
    reviewOutcomes();
    checkNullPointerSnapshots();
    createAssertions();
    checkAssertionResults();
}
```

The tutorial test works great as a soap opera test, but it also finds those subtle changes in the product that cause the documentation to get out of date very quickly. If you have ever tried to keep tutorial documentation up to date, you know how tedious that can be. With an automated test, the writers are notified immediately when a test fails because the product has changed.

comes out, there is probably something wrong.

We had attempted to automate system-level tests before. We started by building a harness so that . . . well, you guessed it, that project was abandoned. This time, we decided to take a page out of the Agile developer's rulebook and build the harness one test at a time.

THE FIRST TEST

My company's product Agitator is a tool for automating, creating, and managing developer tests for Java code. Agitator has a great deal of complexity under the covers, but the basic algorithm for using it is quite simple. First, you agitate a Java class. Agitator then generates observations that describe what the code does, which you turn into assertions that describe what the code should do.

I decided that the first test should agitate a simple Java class that our CTO uses to introduce Agitator in demos and then check the observations.

```
public class SimpleMath {
    public static int add(int x, int y) {
        return x + y;
    }
}
```

When I agitate this class, Agitator observes that the return value of the "**add()**" method always equals x + y. I started writing a test that uses the same steps that I had performed in the product's user interface.

```
public class ObservationSmokeTest extends TestCase {
    public void testXPlusY() {
        agitate(SimpleMath.class);
        OutcomeHarness outcome = getOutcome(SimpleMath.class,
                                            "add(int x, int y)" );
        outcome.assertObservationExists("@RETURN == x + y");
    }
}
```

A quick note on jargon: Agitator generates its observations in the form of Java expressions with a few special keywords like @RETURN, and we use the term "outcome" to refer to the result of a method call.

With the first test, I made several important design decisions for the harness.

The tests will be written in Java using JUnit. This was an appropriate design decision for our team because all the testers know Java very well. In different circumstances, I might have chosen a different technology such as a scripting language or FIT.

The tests will follow the "shape" of the user interface (UI) but will not test through the user interface. For example, I "agitate a class" rather than "select a class in the project tree and click the agitate button." Tests at the UI layer can be very fragile and hard to maintain.

The harness will model the system at a conceptual level that a future tester can associate with parts of the UI. In *The Design of Everyday Things,* Donald Norman tells us that users form a mental image—a conceptual model—of how a system is constructed. In some systems, the interaction designers go to great lengths to present a conceptual model that is quite different from actual implementation. In other systems, the conceptual model and the implementation model may be identical.

The idea that the harness will model the system at the conceptual level is simple but powerful. It will make it easier for future testers to find their way around the tests without needing a deep understanding of the system architecture. If you have spent much time looking at automated tests, you know that they often include so many implementation details that the intent of the test is obscured. I hope to hide all the thorny implementation details inside the test harness. The harness will be thorny, so the tests can be smooth and silky.

At the end of the first round of tests, I expect to have a class

Who Should Build the Test Harness?

Building a test harness is frequently assigned to a specialist toolsmith on the testing team. Sometimes a developer will be pulled away from his day-to-day responsibilities to build a harness. Either of these approaches is a certain recipe for a harness that sits on the shelf. The best approach is for a tester and a developer to implement the first few tests together and get the bulk of the harness done. Even if your testers have the programming skills to build a robust harness, they often will run into roadblocks put in place by the development team who may resent being told to change the design so it can be tested. Having a developer on the team can give the testers the political clout to knock down those roadblocks. It also can make the developer more sensitive to the need to make the product testable in the future. Ideally choose a senior developer, who can offer the political and technical contribution a junior developer may not.

REFACTORING ACROSS BOUNDARIES

Conway's Law says, "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." For example, if you have four groups working on a compiler, you'll get a 4-pass compiler.

If the lines between your development, testing, and tools teams are drawn too rigidly, it can be difficult for the harness to evolve to an optimum architecture. If the testing people have to ask the tools people for every change, cruft will start to build up in the tests. Similarly, if the tools people are not allowed to make changes to the production code, cruft will build up in the harness. If developers, testers, and tools people work together closely, or if they are on the same team, code can migrate seamlessly to the place where it is most appropriate.

in the harness for each concept that would be meaningful to a user of Agitator. Each test should read like a set of instructions that you might give to an expert user over the phone.

"Agitate the simple math class. Is Agitator done? OK, look at the outcome for the "**add()**" method. It should show that the return value equals x+y."

The harness is precisely the code that will allow the tests to speak the language of a user. It enables my test to say:

```
agitate(SimpleMath.class)
```

instead of:

```
Project project = ProjectManager.
                    createProject("target/SmokeTest.arx");
project.setClasspath("target/classes");
ProjectManager.setProject(project);
RunnerIterationThread thread = new ClassRunnerThread(target);
agitator.start();
agitator.join();
```

Checking the results becomes

```
outcome.assertObservationExists("@RETURN == x + y")
```

rather than a bunch of code that rummages through XML files.

Functional Testing is not TDD

I have been practicing Test Driven Development (TDD) for several years now, and I find that the style I use for writing functional tests is quite different from the style I use for unit tests. For one thing, when I write a functional test, the "function" usually is already in place. TDD-style tests are mostly about using tests to drive the design of the code; the resulting tests are of secondary importance.

When I write a functional test, I am very conscious that a future reader of the test may not be a professional programmer, so I make an extra effort to make it readable. I start with a complete test expressed in high-level method calls. Once the test is working, I aggressively remove any incidental code that might obscure the intent of the test.

The goal of the test is different, too. In a unit test, we try to separate the unit from the rest of the system so that we can test it in isolation. By contrast, we want a system-level test, especially a smoke test, to touch as much of the system as is practicable. The system tests complement the unit tests and catch some of the bugs that may fall between the gaps that sometimes appear when separate units are integrated to build the final system.

A well-written functional test can take the place of a requirements document and a test plan—but only if you make the effort to keep the tests readable. Even if the customer would never dream of writing a test in Java, he will usually appreciate being able to read the tests written on his behalf.

The best way to discover what functionality to include in a harness is to start with specific tests and make them work, rather than plan out the grand harness.

A HARNESS APPEARS

I have a test that won't even compile but, with just a few keystrokes and a little magic from my IDE, my test looks like this:

```
public class ObservationSmokeTest extends TestCase {
    public void testXPlusY() {
        agitate(SimpleMath.class);
        OutcomeHarness outcome = getOutcome(SimpleMath.class,
                                            "add(int x, int y)" );
        outcome.assertObservationExists("@RETURN == x + y");
    }

    private void agitate(Class target) { }

    private OutcomeHarness getOutcome(Class target, String method) {
        return new OutcomeHarness();
    }
}
```

OutcomeHarness is the first of many classes that will make up the complete harness. I don't like to leave assertions that pass by default, so I make it fail with a message that reminds me to add the body of the assertion later.

```
public class OutcomeHarness {
    public void assertObservationExists(String observation) {
        Assert.fail("not done yet");
    }
}
```

I now can run the test and, just as I expect, it fails.

The next task is to implement the **agitate()** method. I am not familiar with this part of the codebase, but I remember that there is an action class called AgitateAction and start my investigation there. Actions are method objects that Swing applications use to encapsulate the behavior of UI commands and are often a good place to start looking if you want to know how the system works. I find the code that initiates agitation, and I copy that code into my test.

This time when I run the test, it takes a little longer before it fails, and I can see result files that tell me the agitation happened. The files contain XML, which I am tempted to parse to get the information I need. But, I want to make my test behave more like the user interface, so I find the UI class that displays the outcome results and copy that code into the test harness.

Next, I fill in the details of the OutcomeHarness.

```
public class OutcomeHarness {
    private Outcome outcome;
```

```
    public OutcomeHarness(Outcome outcome) {
        this.outcome = outcome;
    }

    public void assertObservationExists(String observation) {
        List observations = outcome.getAssertionsAndObservations();
        for (Iterator iterator = observations.iterator();
                                    iterator.hasNext();) {
            Object candidate = iterator.next();
            if (observation.equals(candidate.toString())) {
                return;
            }
        }
        Assert.fail("Observation does not exist \"" + observation + "\"");
    }
}
```

Sometimes it's tempting to add query methods to the harness so that the test code can ask for information and make assertions about it. The "tell, don't ask" principle says that client code should tell an object what to do rather than ask for its details and do it itself. Applying this principle, I like to create specialized assertion methods that check the results inside the harness. This encapsulation helps to prevent the implementation details from leaking out into the tests, which can impact readability and result in duplicate code. In this example, if the formatting of the observation ever changes, we can change the existence check to something more sophisticated—and we can do it in one place in the harness rather than in hundreds of individual tests.

I run the test again and it passes, so it is a good time to review—I have one very shallow test, but I have a harness that allows me to test the system from end to end. I did a lot of work for that first test, but that work will pay off as I add more tests.

A SECOND TEST

I want to get broad coverage of every subject area of the product before I go deep into any one subsystem, so I move to a new area. Agitator allows the user to add assertions, which get evaluated on all subsequent runs. Assertions are just Boolean expressions, and the simplest expressions I can think of are "true" and "false." As always, I perform the test manually in the UI first. I add a "true" assertion and a "false" assertion. As I expect, the "true" assertion passes and the "false" assertion fails. I start a new test class for the new subject area.

```
public class AssertionSmokeTest extends TestCase {
    public void testPassAndFail() {
        OutcomeHarness outcome = getOutcome(SimpleMath.class,
                                            "add(int x, int y)" );
        outcome.addAssertion("true");
        outcome.addAssertion("false");

        agitate(SimpleMath.class);

        outcome.assertAssertionPassed("true");
        outcome.assertAssertionFailed("false");
    }
}
```

I realize that the two tests have a lot in common and decide to extract a common superclass. Rather than show all the refactoring steps in code, I'll just show the resulting model as a UML class diagram. (See Figure 1.)

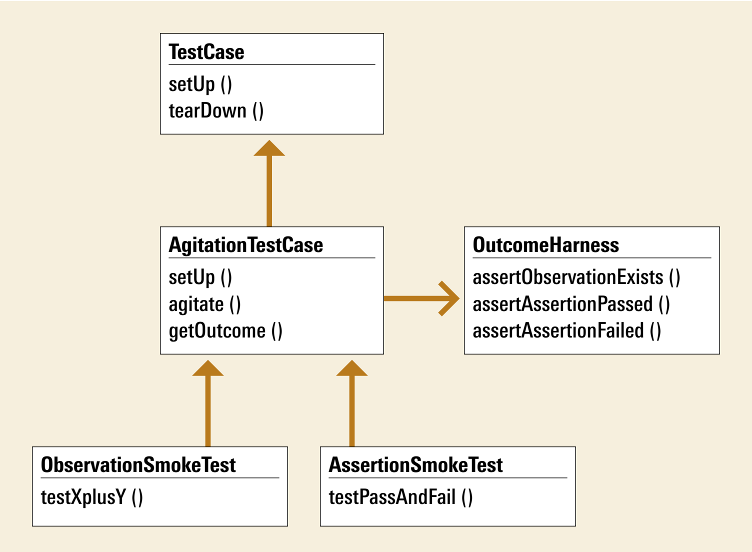

Figure 1: This is the result of refactoring the tests to remove duplication.

THE COMPLETE SMOKE TEST

With the second test done, I continue adding tests for all the other areas of the system. I end up with thirty tests spread across all the major subsystems and a test harness consisting of about twenty classes. As a final step, I create a test suite to make it easy to invoke the whole test from CruiseControl.

```
public class SmokeTest extends TestSuite {
    public static TestSuite suite() {
            TestSuite suite = new TestSuite();

            suite.addTestSuite(AssertionSmokeTest.class);
            suite.addTestSuite(COASmokeTest.class);
            suite.addTestSuite(CoverageSmokeTest.class);
            suite.addTestSuite(EjbSmokeTest.class);
            suite.addTestSuite(KitchenSinkSmokeTest.class);
            suite.addTestSuite(FactorySmokeTest.class);
            suite.addTestSuite(ObservationSmokeTest.class);
            suite.addTestSuite(OutcomeSmokeTest.class);
            suite.addTestSuite(SnapshotSmokeTest.class);
            suite.addTestSuite(TestClassSmokeTest.class);
            suite.addTestSuite(UserExpressionSmokeTest.class);
            suite.addTestSuite(WebSmokeTest.class);

            return suite;
    }
}
```

(Continued on page 45)

(Continued from page 29)

GROW YOUR TEST HARNESS NATURALLY

The entire suite takes two minutes to run and is surprisingly effective at finding regressions. Our unit tests prevent most bugs from getting into the code, but the smoke test usually catches the ones that do make it through. This saves a lot of time that might otherwise be wasted installing brain-dead builds and frees up resources for manual testing and designing more targeted system-level tests.

INCREMENTAL DEVELOPMENT LEADS TO A MORE FLEXIBLE DESIGN

As with every major automation effort, an extensive test harness was needed. Beginning with a modest goal and refactoring as I went along, I was able to construct a harness that was just powerful enough for the task at hand but flexible enough to grow to meet our future testing needs.

This incremental style of development, where you add just enough code to satisfy the immediate requirement, was formalized in Kent Beck's *Test Driven Development* and is rapidly becoming popular in programming circles. In Test Driven Development (TDD), the programmer writes a brief test and then writes just enough code to pass the test. When a subsequent test results in duplication, the programmer refactors and a design emerges through the repeated application of these simple actions.

Developers using Agile processes know that incremental planning and design result in a robust and flexible architecture with less time wasted building features that will never be used. The same ideas applied to automating system tests can quickly deliver valuable, running tests—not just a harness that sits unused on a shelf. **{end}**

Kevin Lawrence works at Agitar Software—a company that shares his passion for software quality. Kevin can be reached via email at kevin@agitar.com.

page 46
full-page ad
SQE: PowerPass