

TELL ME

Got deployment ills? Give your application

WHERE IT

a suite of diagnostic tests and your customers

HURTS

BY MIKE CLARK

may not have to call you in the morning.



MINUTES AGO, YOU RELEASED THE LATEST VERSION OF YOUR BUZZWORD-COMPLIANT HOSTED APPLICATION to your most important customer. It's no small accomplishment for the team and is a potentially huge deal for your company. Your face lights up with a big, toothy grin as a wave of high-fives goes around the team. When the excitement turns into a collective sigh of relief, the team huddles up to depart for a well-deserved celebration dinner. Just then, the sound of a ringing telephone cuts through the air, freezing everyone in their tracks. It's the support phone, and that could only mean one thing: deployment trouble. ►►

Fred, the guy at the customer site responsible for deploying your application, is happier to hear your voice than you are to hear his. He has successfully deployed previous versions of your application, but this time something went terribly wrong. He explains that when he tries to fire up the application, it belches a console full of cryptic messages and then crashes spectacularly. You know it worked on your machine, and it even passed all of the unit and system tests on your QA server before being released to Fred, but that's no consolation to him now. His company is losing money every minute that the new features in this release aren't deployed.

With the clock ticking off dollars, you start to troubleshoot by walking Fred through the multistep deployment procedure again, just to make sure he followed all steps correctly. Meanwhile, the computer on which your application has been deployed sits by idly watching (and giggling) as Fred performs manual checks. Alas, you're both unable to spot any mistakes. "It must be something obvious," you mutter under your breath. "If only we could ask the application what's wrong."

And just as Fred gives in and requests that you drive all the way over to the customer site for a look-see, you remember that indeed you *can* ask the application where it hurts. Wiping the little beads of sweat from your forehead, you ask Fred to type a single command. That command automatically runs a suite of diagnostic tests included with your application—diagnostic tests that are trained to pinpoint deployment problems. A few seconds go by and then you hear Fred blurt out "Aha!" You can already taste that celebration dinner as you ask Fred what the test report says to do next. Fred tells you that the report says that the application is using an incompatible version of the Java Virtual Machine (JVM) and what version the application requires. It's a silly environment problem that would have taken much longer to catch manually, especially when both you and Fred are tired and under pressure. A few keystrokes later, all of the diagnostic tests pass, and the application starts up cleanly.

Fred's an old hand at deploying software, but he's truly impressed that your application comes with these diagnostic tests. He'll use those tests again when his

company buys more licenses to be deployed at multiple sites, if not sooner. Your competitors have a lot of catching up to do. After another round of high-fives, your team swaggers out of the office for a fun night on the town.

THE TROUBLE WITH TROUBLESHOOTING

Unfortunately, too many applications just blow up—gracefully or awkwardly—when they haven't been deployed correctly. Those of us behind the keyboard are left to perform emergency surgery with only a ticking clock to keep us company.

Many software teams try to help users cure deployment ailments by documenting the most common symptoms in a troubleshooting checklist or a list of frequently asked questions (FAQ). Users must hope that the documentation has been kept up to date and, when vexed by a problem, that the documentation will help them find and fix the problem. As a last resort, users might even have the patience to phone up your team for help. You then ask the user to try some things and read back the results. And with any luck you both have an "Aha!" moment before reaching the end of the "secret" checklist.

While these troubleshooting approaches usually work in the end, they are highly inefficient. For starters, you have better things to do than spend countless hours on the phone trying to remotely diagnose deployment problems in the field. Worse yet, the burden of resolving problems is on the user—the person simply trying to deploy your application. Users have better things to do than double check by hand everything that could go wrong. To them, deployment is simply a means to an end: using the application. Having to diagnose problems doesn't exactly give them a good first impression of your application.

AUTOMATED CHECKLISTS

Diagnostic tests travel with your application wherever it goes to make sure it always ends up in a healthy environment.

Once your application has been deployed, the tests feel around the edges of the application to ensure that it has been properly "seated" by the deployment procedure. Just like a printer lets you print a test page before you try to print a novel, software should let users run a suite of diagnostic tests before they attempt to start using it. An application that comes with diagnostic tests gives users confidence that the application is ready to be used.

Say, for example, the following three diagnoses are responsible for the majority of deployment ailments that affect users of your application:

1. An incompatible version of Java is installed.
2. An output directory is specified in a configuration file, but the application doesn't have permissions to write to that directory.
3. The URL used to connect to the database isn't configured properly.

After about the third time you've spent precious hours of your life helping a user find and fix these problems, you're ready for some automation. That is, you're ready to put the computer to work finding the problems and offering suggestions for how to fix them. This is good for you, and it's even better for users. What users really want is to run a troubleshooting checklist and, if the deployment environment is unstable, have the *application* tell them how to fix the problems. Or, better yet, they'd like the troubleshooting checklist to be run automatically as part of the deployment.

WRITING DIAGNOSTIC TESTS

You write diagnostic tests to check an application's external environment in a manner similar to the way you write unit tests to check the application's internal integrity. In both cases you assert, in the form of a test, an expectation that should always be true, and, when that test is run, the computer tells you if your expectation was met. By coding expectations about the deployment environment into

```

import java.io.File;
import java.io.IOException;
import junit.framework.TestCase;

public class DiagnosticTests extends TestCase {

    public void testRequiredJavaVersionIsInstalled() {

        String version = System.getProperty("java.version");

        String message =
            "Incompatible Java version. " +
            "The application requires version 1.4.x, but the " +
            "$JAVA_HOME environment variable refers to version " + version;

        assertTrue(message, version.startsWith("1.4"));
    }

    public void testOutputDirectoryIsWritable() throws IOException {

        Configuration configuration = new Configuration();
        File outputDirectory = configuration.getOutputDirectory();

        String message =
            "Unable to write to the output directory '" +
            outputDirectory.getAbsolutePath() + "'. " +
            "Please ensure the current user has write " +
            "permissions on this directory.";

        assertTrue(message, outputDirectory.canWrite());
    }

    public void testDatabasesAccessible() {

        Database database = new Database();

        String message =
            "Unable to connect to the database '" + database.getURL() + "'. " +
            "Please check the 'database.url' property " +
            "in the 'database.properties' file.";

        try {

            database.connect();

        } catch (Exception exception) {
            fail(message);
        }
    }
}

```

Figure 1: This diagnostic test checks for three common deployment problems.

computer-checked assertions, you get a checklist that can be run accurately and consistently at the push of a button.

An easy way to start writing diagnostic tests is to use your current unit testing tool. JUnit, for example, or any variant of JUnit for languages other than Java, works quite well as a diagnostic testing tool because it executes computer-checked assertions that you write using

the full power of Java. For example, the diagnostic test in Figure 1 is written in JUnit and automatically detects the three deployment problems that cause your users the most grief.

The `testRequiredJavaVersionIsInstalled()` test inspects a system property to verify that a compatible version of Java (version 1.4.x) is installed and was used to run the test. The `testOutputDirectoryIs-`

`Writable()` test checks that the configured output directory exists with write permissions for the user running the test. Both of these tests use the `assertTrue()` method, which takes two parameters: a message to print if the assertion fails and a condition that's expected to return a true value.

The third test, `testDatabasesAccessible()`, attempts to connect to the database. If the connection attempt causes an exception to be raised, then the `fail()` method is called to print out a helpful message that includes the suspect configuration value and the name of the configuration file that contains that value.

At first blush, these diagnostic tests don't appear to do all that much. It's probably not the kind of thorough testing you're used to seeing. Nevertheless, these simple tests detect deployment problems more efficiently than any troubleshooting checklist or phone conversation. Think of them as executable documentation that's always up to date. Users can count on the fact that whenever they deploy your application and it fails to start cleanly, the diagnostic tests will also fail, giving them information about the problem and, more importantly, how to fix it.

MAKING IT SIMPLE

Given how frequently users will deploy new versions of your application, running the diagnostic tests needs to be easy. At a minimum, the person deploying your application should be able to run all of the diagnostic tests in one fell swoop. This implies that the mechanism to run the tests needs to be tailored to the skill level of your application's average user.

If, for example, your application runs in a Web server environment, then consider distributing a diagnostic webpage with a button that when pressed runs all of the tests. One of the advantages of this approach is that the test results can be displayed right there in the browser as nicely formatted HTML. The downside is that the Web application needs to have been successfully deployed to some extent before the diagnostic webpage can be used.

A low-tech solution that doesn't require much infrastructure is to include a batch file or a shell script with your ap-

```
.FFF
Time: 0.103
There were 3 failures:

1) testRequiredJavaVersionIsInstalled(DiagnosticTests)
   junit.framework.AssertionFailedError:
   Incompatible Java version. The application requires version 1.4.x,
   but the $JAVA_HOME environment variable refers to version 1.3.1

2) testOutputDirectoryIsWritable(DiagnosticTests)
   junit.framework.AssertionFailedError:
   Unable to write to the output directory '/path/to/output'.
   Please ensure the current user has write permissions on this directory.

3) testDatabaseIsAccessible(DiagnosticTests)
   junit.framework.AssertionFailedError:
   Unable to connect to the database 'jdbc:mysql://localhost/test'.
   Please check the 'database.url' property in
   the 'database.properties' file.

FAILURES!!!
Tests run: 3, Failures: 3, Errors: 0
```

Figure 2: Failed test results include the test name and diagnosis.

plication. Executing this file runs all of the diagnostic tests. For example, after deploying the application, the user simply types:

```
$ cd /path/to/yourapp
$ diag_test
```

The `diag_test` script could transparently run one or a thousand diagnostic tests. Assuming it runs the three tests in the `DiagnosticTests` test case we wrote earlier and they all fail, the output is printed to the console, as shown in Figure 2.

For each failed test, the user sees both the name of the test and the message that was provided as the first parameter to the JUnit assertions in the `DiagnosticTests` class. The messages are intended to help the user resolve each deployment issue. But are these messages intuitive and detailed enough to help *your* users? The answer depends on their familiarity with the application. The good news is that you'll hear about it if the messages aren't helpful, so be prepared to refine the messages over time. (See the sidebar "Good Bedside Manner.")

Just as it's important to help users fix problems when the tests fail by printing information they can act on, it's equally important to let users know when the ap-

plication is ready to be used. Nothing conveys that better than a simple and unambiguous "OK" message. If all of the diagnostic tests in the `DiagnosticTests` class pass, for example, then the JUnit test runner prints the following confident message to the console:

```
...
Time: 0.103
OK (3 tests)
```

POST-DEPLOYMENT CHECKS

Pre-deployment diagnostic tests don't preclude routine post-deployment checks, of course. For example, the first time a user does something that requires the database, the application will attempt to connect to the database. If the database isn't available, the application will fail, ideally sending the user a friendly error message while recording all of the gory details of the problem in a log file. But why wait for post-deployment checks to find out there is an environment problem? At post-deployment, the user has already invested time in an application that wasn't well seated in the first place. He begins wondering, "What's the next

thing that might go wrong? Am I feeling lucky?"

Once you start writing diagnostic tests, you can use them throughout the life of the application. If the application fails to start altogether, the user can pretest the deployment environment by running all of the diagnostic tests using an external command. If the application starts just fine, an additional safety mechanism can be put in place to programmatically run the diagnostic tests from inside your application, for example, as the first step in the application's `main()` method. This post-deployment check continually ensures that nothing has been changed in the deployment environment.

LESSONS FROM OPEN SOURCE

Perhaps your team builds libraries that are used by deployed applications, rather than building turn-key applications. What do diagnostic tests have to offer you? Well, consider that your library lives within a deployment environment that may become unstable. And if your library has configuration options that need to be tweaked when deployed, then it's susceptible to human error. You can't avoid human error, but you can minimize it with good diagnostic tests. That means that by shipping diagnostic tests with your library, you'll also minimize the number of times your support phone rings.

Taking this a step further, what value might there be in distributing unit tests you used during the development of your software? To help answer that, it's interesting to consider that open source projects are blazing new trails when it comes to testing. Not only has writing unit tests become a compulsory skill on many popular open source projects, but those same projects also distribute all of the unit tests.

Agitar Software recently published quality metrics for several prominent open source Java projects as a way to promote distributing tests and to help open source projects advance their testing efforts. Testing metrics for several projects are summarized in Table 1. We have to be careful not to read too much into those numbers, of course. The

quantity of tests shipped with software is no indication of the software's overall quality. However, the trend of distributing tests with open source projects is compelling.

It would be easy to assume that this trend is due to the distributed nature of open source software development. Multiple developers may be working on the software at any given time and place, and automated tests continually keep their collaborative work in check. But if testing were simply for the benefit of the core development team, then the tests could easily be stripped from the distribution file when an official release is packaged. And yet, increasingly, open source projects are making a conscious decision to include unit tests in official releases.

What value might open source projects be realizing by distributing unit tests that could be equally applicable to commercial projects? Consider the following potential advantages:

- **Improves software quality** When you distribute tests, you allow users of your software to review, and even improve, those tests. Although the number of tests you distribute is unrelated to the overall quality of the software, users feel more confident using your software when it includes open tests that they can extend.

PROJECT	NUMBER OF ASSERTIONS	TEST COVERAGE INDEX	CLASSES WITH TEST POINTS
BerkeleyDB	2875	61.1	48.6%
CruiseControl	804	56.2	54.1%
Lucene	1171	61.0	31.1%
JDOM	1014	38.5	20.3%
Spring	4407	65.6	45.7%

SOURCE: Agitar Software (<http://agitar.com/openquality/>)

Table 1: Test statistics for open source projects

- **Provides example code that works**

The easiest way for other programmers to integrate your library into their applications is by first learning from example code. By distributing tests with your library, you're in effect providing one example client of the library's API. That is, the tests document in an executable format how you intended your API to be used.

- **Encourages accurate bug reporting**

Failed tests are an excellent way to report bugs. If you ship tests with your software, users who are programmers are more inclined to use tests as a means for communicating specific examples of the software not behaving as expected. The test gives you a way to quickly reproduce the bug—and to know when you've squashed it.

Simply by choosing *not* to strip out tests when a distribution file is created, open source projects are providing more value to their users. At the same time, the overall quality of the software improves through contributions from those users. It's a win-win situation that seems perilous to ignore on commercial projects.

That's not to say that every project should ship all of its unit tests. Some tests may be specific to your environment and not portable enough to pass when distributed to the far reaches of the world. And depending on how many unit tests you've written and the amount of test data that goes with them, distributing all of your tests may make the size of the distribution file unwieldy. In those situations, a more pragmatic approach is to distribute handpicked unit tests that exercise code at the boundary where client code interacts with your library.

Back in Fred's world, after correcting the JVM version problem, the application survives a battery of additional tests on the staging server. It's time to promote the application to the production server. Now that Fred knows how to run the diagnostic tests that are included with the application, he doesn't fear pressing the "Return" key that makes the system go live. The production deployment goes off without a hitch, and everyone makes it to dinner on time. **{end}**

Good Bedside Manner

Users don't want to have to email diagnostic test results to you, the all-knowing programmer, to be translated into something they understand. Instead, make users want to read the test results and act on them by carefully wording the corrective action messages in the test results. A good diagnostic message includes at least the following information:

- Clear and concise instructions expressed in the domain vocabulary, when possible
- The name of a configuration value to check and where that value is set (e.g., command line, environment variable, properties file, etc.)
- An example configuration value in the case that the value requires a unique format

When JUnit is used to run diagnostic tests, a stack trace is printed for each failed test. It's useful information to a programmer, but it's unwelcome noise to your average user. This is where the programmer-centric focus of JUnit starts to leak through to the user-centric focus of diagnostic tests. Thankfully, JUnit is a framework that you can easily extend. Rather than using the built-in JUnit test runners, consider writing a custom test runner that prints the test results in a format that's best suited for your application's users.

Mike Clark is the author of Pragmatic Project Automation (published in 2004 by The Pragmatic Bookshelf), editor of PragmaticAutomation.com, and the creator of several popular open source tools. Mike helps teams build better software faster through his company, Clarkware Consulting, Inc. (<http://clarkware.com>). Contact him at mike@clarkware.com.