JUnit + Jtest = Automated Test Case Design and Static Analysis

Jtest, an automated unit testing and static analysis tool, extends JUnit to provide automatic test case generation and static analysis functionality.

by Carson Ellsworth and Cynthia Dunlop

If you currently use JUnit for Java unit testing, you can extend it to automate test case design and coding standard enforcement with Parasoft Jtest. Jtest is an automated unit testing and static analysis tool that includes and extends JUnit to provide automatic test case generation and automatic checking of over 300 Java coding standards.

Test Case Generation Benefits
By extending JUnit with Jtest, you can automatically generate JUnit-compatible test cases that verify code construction, stability, and—if the class uses Eiffel Software's Design by Contract—functionality. This allows you to generate a foundational test suite you can extend with JUnit test classes.

Jtest's automatic test case generation helps you prevent errors in two main ways:

1. It enables you to quickly produce all the required test cases for thorough construction testing, which is impractical to achieve manually. Jtest tries to create test cases that execute every possible branch of each method it tests. For example, if the method contains a conditional statement (such as an if block), Jtest will generate test cases that test the true and false outcomes of the if statement.
2. It enables you to instantly generate and execute test cases as soon as you are done writing or modifying a class. This prompts you to find and fix problems before you (or a team member) unwittingly introduce additional errors by adding code that builds upon or interacts with the problematic code.

Static Analysis Benefits
Jtest's static analysis feature provides additional error-prevention safeguards. During testing, Jtest will statically analyze each class by parsing its Java source code and checking whether it follows a set of over 300 coding guidelines or "rules" designed to identify error-prone code. You can also design and enforce custom coding guidelines as needed. Each time you find an error, you can determine why the error occurred, develop a custom rule that prohibits the cause of this problem, then automatically enforce this rule to ensure that Jtest immediately identifies any repeat offenses.

How to Extend JUnit with Jtest: A Simple Example
To better understand how Jtest can extend JUnit, review a simple example of these two tools working together. Assume you have just written Simple.class, a very...
basic class (see Listing 1 for its source code), and you want to verify whether it is correct and robust before you start working on another class.

If you relied only on JUnit, you might start the testing process by creating a test class like the one shown in Listing 2.

If you compile this test class then execute it using JUnit, all the test cases will pass. You probably would then check the source code into the source code repository and begin working on the next class.

However, if you use both Jtest and JUnit in this scenario, you would perform the following steps:

1. Load Simple.class into the Jtest Class Testing UI.
2. Click the Start tool bar button to start the test (as shown in Figure 1).

Jtest examines the class under test, statically analyzes the class's source code, designs test cases that test the class's construction and attempt to cover every branch of every method, and then executes both the automatically generated test cases and the six JUnit test cases represented in the SimpleTest test class. This entire process takes less than 25 seconds. If the class contained Design by Contract comments, Jtest would have also designed test cases that verify the class's functionality.

When the test is completed, Jtest reports the results shown in Figure 2.

By using Jtest with JUnit, you identified 12 violations of Java coding guidelines and one uncaught runtime exception. In addition, you gained 15 new test cases that you can export as a JUnit test class and/or use for functionality testing.
What Jtest Caught

To get an idea of the types of problems that Jtest identifies automatically, let's examine the uncaught runtime exception and one of the static analysis violations that Jtest uncovered. The uncaught runtime exception message shown in Figure 3 reveals that the `startsWith` method is implemented incorrectly. The method should return `false` for the argument `""` and `"0"`, but it throws a runtime exception instead. If the error is not fixed, any application using this class will eventually crash or give incorrect results. By identifying and fixing this problem immediately, you can prevent significant grief later.

![Figure 3](A Runtime Exception that Jtest Exposed)

The violation of the PB.TLS static analysis rule shown in Figure 4 reveals that the developer of this class inadvertently wrote `case10` instead of `case 10`. If the class is not fixed, it will give incorrect results when it is passed the value 10. If this problem is allowed to remain in the code, it will likely cause strange, difficult-to-diagnose errors during application testing or in the field. Once again, by identifying and fixing this problem immediately, you can prevent grief later.

![Figure 4](A Coding Standard Violation that Jtest Identified)

Create a JUnit-compatible Test Class for Automatically Generated Test Cases

Finally, assume that you want to create a JUnit test class that represents the 15 automatically generated test cases. You choose Jtest's Export to a JUnit Test Class option. Jtest instantly generates a test class that allows you to run these 15 test cases in either Jtest or JUnit. You can add more functional test cases by modifying the test methods in the automatically generated test class, and then you can run the modified test class in either Jtest or JUnit. Adding test cases in this manner enables you to verify whether the class is correct. This strategy significantly reduces the amount of code that you need to write for each test—without sacrificing test quality.

Integrate Unit Testing into Daily Development

These days, almost every Java developer recognizes the value of practices such as unit testing and coding standard enforcement, and intends to perform them as part of his or her regular testing practices. However, these practices can be time-consuming, and most developers rarely have the opportunity to perform them as early, frequently, or thoroughly as they would like. Extending JUnit with Jtest can solve this problem. When you use both Jtest and JUnit, you can realistically integrate unit testing and coding standard enforcement into your daily development practices.

Carson Ellsworth is a Lead Systems Engineer for Parasoft Corporation as well as Java product support manager. He has extensive knowledge of Java design and testing paradigms. His professional experience includes software engineering, design, documentation, and quality assurance. You can reach him at carson@parasoft.com.

Cynthia Dunlop has been responsible for crafting Parasoft product manuals and technical papers since 1998. Dunlop has also authored numerous technical articles about issues related to software development. You can reach her at cynthia@parasoft.com.