Automating Java Unit Testing with Jtest
Parasoft Jtest, an Automated Error Prevention product that automates Java unit testing and coding standard compliance, is the world’s first product to make team-wide Java unit testing and coding standard compliance practical. Using patented technologies, it has:

- Automated intelligent Java unit test case generation since 1997.
- Integrated automated unit testing and coding standard analysis into a single solution since 1999.
- Automated intelligent JUnit test case generation since 2002.
- Supported team-wide automated unit testing and coding standard compliance since 2003.

Over the past 6 years, it has been recognized with numerous awards—including the prestigious Jolt Product Excellence Award—and been adopted by tens of thousands of developers worldwide.

To help developers produce reliable code in record time, Jtest analyzes classes, then generates and executes JUnit-format test cases designed to achieve maximum coverage, expose uncaught runtime exceptions, and verify requirements that were expressed using Design by Contract. In addition, Jtest automatically records the status of the test execution; if the results are verified, the automatically-generated test cases become functional regression test cases. Additional tests can be added by extending the generated test cases and using new or legacy JUnit test cases. Jtest is the only tool in the industry that performs unit testing at this level. Jtest also checks whether code follows over 400 coding standard rules (plus any number of custom rules) and automatically corrects violations of over 200 coding standard rules. With the click of a button, developers can identify and prevent problems such as uncaught runtime exceptions, functional errors, memory leaks, performance problems, and security vulnerabilities.

In addition, Jtest can standardize and manage Java error prevention across a team and support the Parasoft AEP Methodology, a new methodology for improving software quality and reliability in a team environment. Implementing Jtest as part of a team-wide AEP solution is a critical step in improving software quality and streamlining a team’s software development process.

This paper features Jtest's unit testing capabilities. The nature and value of Jtest's coding standard analysis capabilities are already well-understood and have been described in reviews and other contexts. The examples provided in this paper also showcase how Jtest handles difficult unit testing situations, such as:

- Regular expressions
- Methods that take interfaces as arguments
- Lists
- Abstract classes

**Overview of Jtest's Automated Unit Testing**

Jtest has been automating intelligent unit test case generation since 1997. Jtest can analyze any type of Java class, then generate JUnit test cases that verify the class. Automatically-generated test cases are designed to expose uncaught runtime exceptions and attempt to cover every branch of every method. By running these test cases in Jtest, you verify class robustness and expose inputs that could cause the program to enter an inconsistent state or terminate. If you use Design by Contract (DbC) to add specification information to your code, Jtest automatically creates and executes test cases that verify whether the code functions as specified. If your class references external resources, Jtest adds stubs to the test cases. If previously-tested code changes, Jtest modifies the test cases as needed.

Originally, Jtest created only test cases designed to expose construction errors (such as uncaught runtime exceptions) and saved these test cases in an internal format. Customer after customer remarked that they liked how Jtest's automated test creation helped them pinpoint construction weaknesses and crash-causing errors with no user intervention, but they also wanted an easy way to add substantial user-defined functional test cases—test cases that verify the functionality requirements listed in the specification. Para-
soft responded to this feedback by providing customers two ways to streamline the functional testing process.

First, to respond to customers' requests that we automate the nearly-impossible-to-automate task of creating functional test cases, we extended Jtest with technology that could automatically understand and verify users' functionality requirements. To automate functional testing with Jtest, users only need to document the program's expected behavior using special Javadoc tags as they write their code. These Javadoc tags should define the code's implicit contracts—for instance, conditions that the client must meet before a method is invoked, conditions that a method must meet after it executes, and invariants that a method must satisfy at specific points of its execution—and be written as specified by the Design by Contract (DbC) language. To help users add contracts as effortlessly as possible, Jtest can check DbC coding standard rules during coding standard analysis, report what types of comments are missing and where each comment should be added, then help users add the necessary comments to the code through the Jtest Quick Fix feature. Once the comments are added, Jtest creates functional test cases that verify whether the specified requirements are satisfied at the unit level.

Moreover, with Jtest's companion tool Jcontract, these contracts can also be leveraged to verify the functionality requirements during integration testing. For software to be reliable, it not only needs to work correctly at the unit level, but also at the application level, after the units are integrated. It is very common that code operates properly during unit testing, but fails during integration testing. These contracts are assets that allow teams to verify the code's functionality during integration testing. The benefit of testing with contracts instead of with traditional assertions is that contracts are only activated during testing. The contracts do not slow program performance during production. Jtest and Jcontract can add the contracts for testing, and production code can remain unaffected by the contracts.

Second, to respond to the overwhelming number of JUnit-devotees' requests for help creating JUnit test cases for "real functional testing," we modified Jtest to create its test cases in JUnit format and to support a team's existing JUnit test cases. Users can have Jtest automatically generate and execute test cases to pinpoint construction and (if the user uses DbC) functionality errors during its initial test run. Then, users can easily extend these JUnit test cases into the complex unit, sub-module, module, and application test cases that the JUnit framework supports—the type of complex test cases that are required to truly verify whether the application works as expected. Because the tests are written in the industry-standard JUnit format, most Java developers can instantly understand and extend the JUnit test cases with no additional learning curve. Moreover, thanks to JUnit's flexibility, these tests can be extended to verify everything from public interfaces, to the interaction of several EJBs, to the interaction between a module and a database. The team's cumulative test suite—including unmodified automatically-generated test cases, extended/added test cases, and legacy JUnit test cases—can then be executed in Jtest, monitored for coverage, used for regression testing, and even used for memory leak detection.

In addition to requesting enhanced functional testing capabilities, users also requested features, training, and infrastructure components that would help them standardize and manage unit testing, along with coding standard analysis, across their teams. That's why Jtest now provides extensive support for team-wide Java error prevention. It now supports the Parasoft Team Configuration Manager, which manages group behavior by standardizing settings across the team and automatically managing the sharing and updating of test assets, such as test cases, test settings, and rule files. Jtest Server Edition can test the entire project during the automated nightly build. It offers the team a rich reporting capability; it identifies which developer is responsible for each error and automatically generates comprehensive reports for managers and focused reports tailored for each developer. Moreover, to help the team leverage Jtest's findings, Jtest also integrates with the Parasoft Group Reporting System. Using GRS's dashboard and reports, a team gains unprecedented visibility into the quality, scope, and progress of their complete projects—not only their Java code, but also any related Web service functionality, Web application functionality, database functionality, and so forth.

These enhancements have propelled Jtest from being a highly-praised (yet sometimes daunting) technology into a product that developers worldwide use on a daily basis. Jtest is now extremely easy to use. No matter how developers are creating and storing their projects, they can start testing them in seconds, and
start identifying coding problems during the initial test—with just one click of a button. Developers can automatically correct many errors found. Moreover, because Jtest fully supports and extends JUnit testing, virtually any Java developer can immediately understand Jtest's automatically-generated test cases and start extending them to verify whether code satisfies advanced functional requirements. Parasoft also offers training courses that teach developers a proven methodology for developing effective functional test suites as efficiently as possible.

**Demonstration of Jtest’s Automated Unit Testing**

This section demonstrates Jtest's unit testing capabilities in two examples, and features the Jtest usage process which most Jtest users prefer. We recommend that you follow this process in your own usage of Jtest.

**Example 1: Unit Testing Overview**

For the first example, assume that you are writing a bank account processing program, and you have just completed the following classes, whose source code is provided in Appendix A:

- ITransaction.java: The base interface for all transactions. Note the @post condition in the 'fee()' method that specifies that all transactions should charge a zero or positive fee.
- FreeTransaction.java: The base abstract class for free transactions. It overrides the 'fee()' method.
- Account.java: A class that represents a bank account.
- Bank.java: A class that keeps all bank accounts and has a couple of methods to show operations that can be done.

Some of these classes document functionality requirements with DbC; others do not. By testing a package that includes both documented and undocumented classes, you will see how Jtest performs unit testing in both situations.

**Run the Initial Test**

For an immediate assessment of your classes’ construction and functionality, you can simply load them into Jtest, select the examples.nbank package, then click Jtest's Play button.

When you click the Play button, Jtest automatically performs the following tasks:

- Identifies what code to test (i.e., it determines which code meets your criteria for test scope, which can filter files or lines based on modification date, author, etc.).
- Performs a code review that checks industry-accepted coding standards and/or custom coding standards which verify project or team-specific requirements.
- Generates test cases in standard JUnit test class format. Each test case includes both input to the method and an assert for the outcomes.
- Executes the generated test case, checking DbC comments and calculating real coverage.
- Assigns any coding standard or unit testing errors to the appropriate user (according to the @author Javadoc tags and/or CVS history information). It not only reports what problems were introduced, but also which developer is responsible for each problem.

Also, Jtest can optionally:

- Generate HTML/XML reports and store results for later analysis.
- Add/update the modified test classes to source control.
- Detect memory leaks.
Jtest reports that it generated 10 test cases and achieved 100% coverage during this test. Jtest actually created and executed many more test cases, but it only reports and saves test cases that increase coverage or expose unique errors.

These test cases exposed 3 unit testing errors:

- 2 NullPointerExceptions.
- 1 specification error: the 'FreeTransaction.fee()' method was returning a negative fee.
If you had enabled memory leak detection for this test, Jtest would have also found the following memory leak in the Account class.

When you enable leak detection, Jtest monitors object allocate events and object free events during unit test execution. An object is considered to be leaked if it is allocated but not freed during garbage collection at the end of the unit test.

Fix Errors Found

First, let's look at the reported NullPointerExceptions. When Jtest automatically generates test cases, it attempts to exercise each method with a wide variety of permissible inputs and then alert users to the potential runtime exceptions that could occur with those inputs. It determines what inputs are permissible by reading the precondition requirements documented using DbC. If no input restrictions are documented with DbC, Jtest assumes that any possible method input—including a null input—is allowed. Jtest reports any exceptions that can occur with a permissible input, except for expected exceptions that are documented using the @exception tag/@throws tags or a "throws" clause in the method definition.

As a result, if you assume that certain methods will never receive null inputs but you do not document these assumptions with the @pre tag, Jtest may generate test cases with null method inputs. If these test cases result in NullPointerExceptions, Jtest will report the exceptions because it does not know that the methods implicitly assume that they will not receive null inputs. Note that if you never want Jtest to use null inputs, you simply enable the related test parameter.

If a null input should not be passed to these methods, you would use the @pre DbC tag to specify that null inputs are not permissible. You could add these tags manually, or you could have the Jtest Quick Fix feature add them automatically.
For instance, the Quick Fix for the NullPointerException reported for the Account class adds the following comment above the apply method.

```java
/**
 * @pre transaction != null
 */
```

This not only tells Jtest to refrain from creating test inputs that violate this precondition, but it also makes the code easier to understand and maintain. Moreover, Jtest (or its companion tool Jcontract) will automatically alert you if this method is passed prohibited inputs during unit testing or integration testing.

The other error reported—that the 'FreeTransaction.fee()' method was returning a negative fee—indicates a problem in the class implementation. To correct this problem, you need to manually modify the 'FreeTransaction.fee()' method so it returns 0. You can perform this modification inside the Jtest editor or in your preferred IDE.

At this point, you can click the Play button again to retest the package and verify that the errors have been corrected.
Review the Automatically-Generated Test Cases

At this point, most users like to jump start their functional testing by reviewing the automatically-generated test cases. However, this step is completely optional, and Jtest is useful even if this step is not performed. In that case, Jtest will assume that the current outputs are correct, save a snapshot of the class's current behavior (in the form of JUnit test cases), and alert you if this behavior changes.

If you choose to review the automatically-generated test cases, you tell Jtest whether they produced the correct outcomes and—if not—specify the correct outcomes. Test case information is presented in a simple tree format for fast review, and outcomes can be verified in seconds with user-friendly shortcut menus. When the classes are retested after test case validation, Jtest will report these test cases as "failed" if and only if the given inputs do not produce the correct outputs.

The verified test cases will be treated specially by Jtest, and they will not be overwritten—even if you ask Jtest to regenerate test cases. They will become a permanent part of your functional regression test suite, where they are used to verify that modifications do not introduce problems.

You can now continue working on your code. Because Jtest's test cases and outcomes are stored in the form of normal .java code, the test cases will be refactored along with the code whenever classes/methods are renamed, moved, etc.

Continue Working on Code

Now, assume that you are modifying the Account class. In addition to adding a 'clear()' method, you introduce a bug by changing "-=" to "=" in 'deposit()'. The modified source code is shown in Appendix A.

Retest the Modified Code
To verify the modified code, you simply click the Play button again. Jtest now reports a specification error for the bug that was introduced and adds one test case to the existing JUnit test class for the 'clear()' method.

Note that when Jtest retested this code, it automatically detected the addition of the 'clear()' method, and automatically created a test case for it. You do not have to tell Jtest to recreate test cases or alert it to the code modifications.

Also, note that Jtest does not need to recreate test cases in this situation. Jtest automatically monitors code modifications, and updates and refactors the test suite as needed. For instance, if new methods are added to the code, new test cases are added automatically. If methods are removed, obsolete unverified test cases are deleted. If code is modified in a way that should not affect the test suite, Jtest understands this and does not try to recreate new test cases; instead, it reruns the existing functional regression test suite so that it can alert you if your modifications cause the class behavior to change. Jtest works closely with the source control system, so it not only recognizes what classes were modified, but also what specific lines were modified. As a result, it is sensitive to code changes, but does not recreate its test suite needlessly, which wastes resources and reduces the effectiveness of your regression testing.

**Extend the Test Cases to Test Additional Functionality**

Verifying and correcting the outcomes of the automatically-generated test cases is a good way to begin functional testing. However, to perform true functional testing, developers need to extend the test suite into
a JUnit functional test suite that effectively verifies whether code works as designed. This procedure cannot be automated by validating test case outcomes. Rather, it involves the following process, which requires human intelligence:

1. Identify use cases that cover all actions which your program should be able to perform.
2. Identify the code's entry points—central pieces of code that exercise the functionality that the code as a whole is designed to undertake.
3. Pair entry points with the use cases that they implement.
4. Create test cases by applying the initialize-work-check procedure.
5. Develop runtime event diagrams and use them to facilitate testing.

For details on this process and how it can be streamlined by extending Jtest's automatically-generated JUnit test cases, see "Leveraging Jtest: Best Practices for Using Jtest on Your Code" at http://www.parasoft.com/jtest.

In summary, even Jtest users who do not use DbC can create functional test cases without reviewing test cases. Jtest generates intelligent test cases, and assumes that the current behavior is correct unless you indicate otherwise. In subsequent test runs, Jtest will report an error if the outcome for a method changes. If this occurs, you should evaluate the error and determine if you introduced a bug, if the previous behavior of the program was incorrect, or if the outcome should be ignored.

Example 2: Focusing on Unit Testing with Design by Contract

Next, let's take a closer look at how Jtest leverages DbC to automate the functional unit testing process.

In this example, assume that you have added a new method to the Account class from the previous example and you have attempted to document the method's functional requirements using DbC. For example, to express the requirement "customers enter the Gold Club if the account balance is over $15,000" in DbC, you would add the comment

```
* @invariant isGold () == _balance >= 15000
```

The complete source code for the modified Account class is shown in Appendix B.

First, to verify whether you have added sufficient DbC comments, you can use Jtest's coding standard analysis feature to check whether the class follows DbC rules. Jtest reports that the 'apply()' method violates the rule that public methods should have @post contracts. It also gives you the option to add the @post contract framework through the Quick Fix feature.
Once you have verified that the code contains appropriate contracts, you can run Jtest—again, by simply clicking the Play button. This time, Jtest automatically generates test cases that verify the requirements which were expressed using DbC.

The automatically-generated test cases expose the following error in the code:

In other words, the code implementation did not satisfy the functional requirement "customers enter the Gold Club if the account balance is over $15,000."
Note that this error was exposed completely automatically. You did not need to review any test cases to learn that the program does not satisfy this functional requirement.

From the reported error, you can open the JUnit test case that exposed the error:

```java
/**
 * Test for method: withdraw(int)
 * @see Account#withdraw(int)
 * @author Jtest
 */
public void testWithdraw3() throws Throwable {
    Account THIS = new Account("a phrase", 15001);
    // jtest_tested_method
    int RETVAL = THIS.withdraw(1); // @invariant contract violation
}
```

In sum, with Jtest, much functional testing can be performed automatically if you document the expected behavior of the program using a formal specification language (DbC). This specification is explicit in the .java file and becomes part of the program's Javadoc documentation that the program's consumers can view. If you change projects, or simply forget how code was supposed to work, its requirements are always readily available and clearly expressed.

Moreover, when the code's requirements are expressed in DbC, they can not only be automatically verified at the unit level, but also monitored during integration testing. Unit testing prevents many problems from reaching the application level. However, due to the complexities in ensuring the flawless interaction of many units— written at different times and often by different developers— it is critical that unit testing be followed by integration testing.

When this class is instrumented with Jcontract and run by QA during their regular integration testing, these same functional requirements will be verified automatically. If any other classes cause the functional requirements to be violated, an error will be reported. Essentially, using DbC in this manner carries functional testing efforts from development to QA. Development adds the contracts and uses them to automate unit-level functional testing. Later, when QA performs manual testing and runs functional regression tests, these same contracts will be verified— with no additional effort.
Obtaining Additional Information

Jtest is available now at http://www.parasoft.com. To learn more about how Jtest and other Parasoft solutions can help your organization prevent errors, contact Parasoft today or visit http://www.parasoft.com.

In addition, Parasoft packages customizable solutions that provide a framework for applying Jtest as part of a team-wide, full-lifecycle error prevention strategy. Based on the Parasoft AEP Methodology introduced in this paper, these solutions ensure consistent and uniform behavior across development teams by providing infrastructure, technologies, configuration, and training to support full-team operation. For details on these solutions, contact Parasoft.

Contacting Parasoft

USA
101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe
France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (020) 8263 2827
Germany: Tel: +49 7805 956 960
Email: info-europe@parasoft.com

Asia
Tel: +886 2 6636-8090
Email: info-psa@parasoft.com
Appendix A: Source Code for Example 1

Note

Comments within the classes highlight how Jtest operates Jtest’s support for various difficult-to-test scenarios:

- Jtest supports regular expressions; see the ‘Account()’ constructor for details.
- Jtest can test methods that take interfaces as arguments (Jtest generates stubs); see the ‘Account.apply()’ method for details.
- Jtest can generate inputs for Lists (Jtest generates stubs); see ‘Bank.closeAccounts()’.
- Jtest can test abstract classes (Jtest creates a default implementation for them). See FreeTransaction.java for details.

Account

package examples.nbank;

import java.util.ArrayList;
import java.util.List;

public class Account {
    public Account (String name, int initial_balance) {
        if (name == null || !name.matches ("([a-zA-Z])+( [a-zA-Z]\.)? \[a-zA-Z\]+")
            throw new IllegalArgumentException ("Invalid name: " + name);
        if (initial_balance < 0)
            throw new IllegalArgumentException ("Invalid initial balance: " + initial_balance);
        _name = name;
        _balance = initial_balance;
        _newAccountsLog.add ("name = " + name + ", initial_balance = " + initial_balance);
    }

    public String getName () {
        return _name;
    }

    public int getBalance () {
        return _balance;
    }

    public int deposit (int amount) {
        return _balance -= amount;
    }

    public int withdraw (int amount) {
        return _balance -= amount;
    }

    public void apply (ITransaction transaction) {
        if (transaction.apply (this))
            _balance -= transaction.fee ();
    }

    private String _name;
    private int _balance;
}
private static List _newAccountsLog = new ArrayList ();

Account (modified)

package examples.nbank;

import java.util.ArrayList;
import java.util.List;

public class Account {
    public Account (String name, int initial_balance) {
        if (name == null || !name.matches ("([a-zA-Z]+)\s?([a-zA-Z]+)\s?([a-zA-Z]+)\s?([a-zA-Z]+)\s?")
            throw new IllegalArgumentException ("Invalid name: " + name);
        if (initial_balance < 0)
            throw new IllegalArgumentException ("Invalid initial balance: " + initial_balance);
        _name = name;
        _balance = initial_balance;
        _newAccountsLog.add ("name = " + name + ", initial_balance = " + initial_balance);
    }
    public String getName () {
        return _name;
    }
    public int getBalance () {
        return _balance;
    }
    public int deposit (int amount) {
        return _balance = amount;
    }
    public int withdraw (int amount) {
        return _balance -= amount;
    }
    public void clear () {
        _balance = 0;
    }
    /**
     * @pre transaction != null
     */
    public void apply (ITransaction transaction) {
        if (transaction.apply (this))
            _balance -= transaction.fee ();
    }
    private String _name;
    private int _balance;
    private static List _newAccountsLog = new ArrayList ();
}

Bank

package examples.nbank;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Bank {

    public void addAccount (Account account) {
        //NOTE: NullPointerException reported and quickfix
        _accounts.put (account.getName (), account);
    }

    public void closeAccounts (List list) {
        //NOTE: input for a List collection generated
        int size = (list != null)? list.size () : 0;
        for (int i = 0; i < size; i++) {
            Object element = list.get (i);
            if (!(element instanceof Account))
                continue;
            _accounts.remove ((Account) element);
        }
    }

    private Map _accounts = new HashMap ();
}

FreeTransaction
package examples.nbank;
// NOTE: shows that Jtest can test for abstract classes
public abstract class FreeTransaction implements ITransaction {
    public int fee () {
        //NOTE: incorrect value returned, will be automatically detected
        // because it violates the precondition declared in the interface
        return -100;
    }
}

ITransaction
package examples.nbank;
public interface ITransaction {
    /** @post $result >= 0 */
    int fee ();
    boolean apply (Account account);
}
Appendix B: Source Code for Example 2

Account

package examples.nbank;

import java.util.ArrayList;
import java.util.List;

/**
 * @invariant getBalance () >= 15000 == isGold ()
 */
public class Account {
    public Account (String name, int initial_balance) {
        if (name == null || !name.matches("([a-zA-Z])+( [a-zA-Z]\.)? [a-zA-Z]+"))
            throw new IllegalArgumentException("Invalid name: " + name);
        if (initial_balance < 0)
            throw new IllegalArgumentException("Invalid initial balance: " + initial_balance);
        _name = name;
        _balance = initial_balance;
        _newAccountsLog.add("name = " + name + ", initial_balance = " + initial_balance);
    }

    public String getName () {
        return _name;
    }

    public int getBalance () {
        return _balance;
    }

    public int deposit (int amount) {
        return _balance = amount;
    }

    public int withdraw (int amount) {
        return _balance -= amount;
    }

    public boolean isGold () {
        return _balance > 15000;
    }

    public void clear () {
        _balance = 0;
    }

    /**
     * @pre transaction != null
     */
    public void apply (ITransaction transaction) {
        if (transaction.apply (this))
            _balance -= transaction.fee ();
    }

    private String _name;
    private int _balance;
    private static List _newAccountsLog = new ArrayList ();
}