Getting Started Guide
Introduction

Welcome to Jtest, an automated error prevention tool that tests any Java class or JSP without requiring you to write a single test case, harness, or stub.

With the click of a button, Jtest automatically creates and executes test cases that verify class/JSP construction. If you use Design by Contract (DbC) to add specification information to your code, Jtest also automatically creates and executes test cases that verify whether classes function as specified. Jtest helps you create additional user-defined test cases, and performs automated regression testing with both automatically-generated and user-defined test cases. In addition, Jtest prevents errors with a customizable static analysis feature that automatically checks over 300 industry-respected coding standards and any number of custom coding standards you design with its RuleWizard feature.

Jtest complements and extends JUnit. Jtest not only runs JUnit test cases, but also automatically designs and executes additional test cases that verify the code and increase test coverage. Moreover, Jtest automatically creates JUnit test class templates into which you can easily enter test cases, and exports all Jtest test cases as JUnit-compatible test classes. It even enforces best practices for JUnit test classes during static analysis.

Jtest’s unique test case generation and static analysis technology helps you prevent problems, catch existing problems as early as possible, achieve the fullest possible coverage of the methods, and uncover problems that other types of testing are unable to detect. When you use Jtest to test each class as soon as you compile it, you will improve software reliability while you reduce development time, effort, and cost.

The testing techniques described in this guide provide a quick introduction to Jtest. To learn more about the various ways that you can use and customize Jtest, see the following resources:

- The Jtest User’s Guide, which provides a complete description of all features and customization options. You can access the Jtest User’s Guide by choosing Help> Contents in either Jtest UI.

- The Jtest Tutorial, which provides step-by-step instructions on performing common tasks using sample files. The tutorial lessons cover white-box testing, black-box testing with Design by Contract, adding and executing user-defined test cases and stubs, regression testing, project testing, using Jtest with JUnit, and using Jtest in a group environment. To access the Jtest Tutorial, choose Help> Tutorial in either Jtest UI.

- The RuleWizard User’s Guide, which provides information on how to create, enforce, and enable/disable custom static analysis rules. To access RuleWizard, right-click the Rules tool bar button, then choose Launch RuleWizard from the shortcut menu that opens. You can then open the RuleWizard User’s Guide by choosing Help> View in the RuleWizard UI.
Windows Installation, Licensing, and Startup

Prerequisites
- Windows NT, Windows 2000, or Windows XP
- JDK 1.3 or higher

Installing Jtest
To install Jtest:
1. Run the setup executable that you downloaded from the Parasoft Web site or that is on your CD.
2. Follow the installation program’s onscreen directions. The installation program will automatically install Jtest on your system.

Starting Jtest
To launch Jtest, double-click the Jtest desktop icon.
A Jtest license must be installed before you can begin using Jtest. Contact your Parasoft representative to obtain a license.
UNIX Installation, Licensing, and Startup

Glossary
<jttest-home>: The Jtest installation directory (the directory where Jtest is installed).
<arch>: The platform on which Jtest will be run. For example, solaris, linux, etc..
<compression-scheme>: The compression scheme used to create the Jtest installation archive. ".Z (compressed)" is standard. ".gz (gzipped)" is faster and smaller, but not common.

Prerequisites
- JDK 1.3.1
  - One of the following platforms:
    - Solaris 7 or 8. All relevant patches from Sun that will allow the machine to run the interpreter from JDK 1.3.1 must be installed.
    - RedHat Linux 6.1, 7.1, or 7.2
    - SuSE Linux 7.2
    - Mandrake Linux 8.1

Installing Jtest
1. Copy the jtest.<arch>.tar.<compression-scheme> file to the directory where you would like to install Jtest.
2. Extract the archive using one of the commands listed below.
   - For .gz files, enter:
     
     gzip -dc jtest.<arch>.tar.gz | tar xvf -
   - For .Z files, enter:
     
     uncompress -c jtest.<arch>.tar.Z | tar xvf -
   - Remember to substitute your specific architecture name (for example, solaris, linux, etc.) for <arch>. For example, to extract a Linux .gz file, you would enter the following command:
     
     gzip -dc jtest.linux.tar.gz | tar xvf -

During extraction, a directory named jtest will be created; this directory will contain the program files needed to run Jtest.

Setting the Environment
After you install Jtest, you must set your environment before you can run Jtest. To set the environment:
1. Use the provided shell script to set up your environment or set the environment by hand.
   - To use the script:
     - For bash or sh shells: Run the jtvars.sh script in <jttest-home>.
       For example:
$ cd <jtest-home>
$ . jtvars.sh

- For csh, tcsh, or ksh shells: Source the `jtvars` script in `<jtest-home>`.
  
  For example:
  $ cd <jtest-home>
  $ source jtvars

- To determine which shell you are using, enter:
  $ echo $SHELL

- To set the environment by hand, add the `<jtest-home>/bin` directory to the PATH
  environment variable and add the `<jtest-home>/lib` directory to the
  LD_LIBRARY_PATH environment variable.

2. Add the Sun Microsystems `javac` compiler to your path (if it is not already there).
   Jtest requires the `javac` Compiler for Design by Contract and black-box testing. If you do not
   have `javac` on your shell's path, set the PARASOFT_JDK_HOME environment variable to
   the location of the Sun Microsystems JDK on your machine.

   - `bash` or `sh` shell example:
     $ export PARASOFT_JDK_HOME=/usr/java/jdk1.3.1
   - `tcsh`, `csh` or `ksh` shell example:
     $ setenv PARASOFT_JDK_HOME /usr/java/jdk1.3.1
   - **Note**: If you add to your environment the `bin` directory of the Sun JDK, you do not
     need to set PARASOFT_JDK_HOME.

3. Make your changes to LD_LIBRARY_PATH, PATH and PARASOFT_JDK_HOME permanent.
   To make the changes environment variables, edit your shell's login script. Add the definition
   of the PARASOFT_JDK_HOME environment variable to your login script only if you don't
   have `javac` on your PATH.
   You might want to ask sysadmin for help with this task. Until sysadmin responds, use the
   scripts provided in the `<jtest-home>` directory.

### Starting Jtest

After you have set the environment, you can start Jtest by running the `jtestgui` command.

A Jtest license must be installed before you can begin using Jtest. A Jtest license must be installed
before you can begin using Jtest. Contact your Parasoft representative to obtain a license.

### If Jtest Cannot Locate Your JDK

If Jtest opens a dialog box that asks you to set PARASOFT_JDK_HOME, it is indicating that it could
not find the `javac` compiler required for black-box testing and Design by Contract.

There are two ways to solve this:

- Set the variable to the installation directory of the JDK.
  - `sh` and `bash` shell example:
    $ export PARASOFT_JDK_HOME=/usr/java/jdk1.3.1
  - `tcsh`, `csh`, and `ksh` shell example:
    $ setenv PARASOFT_JDK_HOME /usr/java/jdk1.3.1

- Add the JDK's `bin` directory to the PATH.
• **sh and bash shell example:**
  $ export PATH=$PATH:/usr/java/jdk1.3.1/bin

• **tcsh, csh, and ksh shell example:**
  $ set path=(path /usr/java/jdk1.3.1/bin)
  $ rehash
Testing with Jtest

The testing techniques described below provide a quick introduction to Jtest. To learn more about the various ways that you can use and customize Jtest, see the resources described in this guide’s introduction.

Testing Prerequisites

General Prerequisites

You must satisfy all of the following requirements in order to use the basic Jtest functionality:

- The `.class` or `.jsp` files for the classes you want to test must be available. A `.class` file is a compiled Java source. Without a `.class` or `.jsp` file, Jtest will not be able to perform any tests.
- The `.class` files must be in a directory hierarchy that reflects the structure of the package, regardless of whether they are in jar files, zip files, or in the file system.
- The classes or JSPs referenced by the tested `.class` and `.jsp` files must be available to Jtest.
- If the `.class` or `.jsp` files are in directories, `.zip` files, or `.jar` files, the `.class` or `.jsp` files must be accessible by Jtest.

To use full Jtest functionality (static analysis, source browsing, Design by Contract, etc.) the `.java` source or `.jsp` files must be available to Jtest during testing.

You can set CLASSPATH parameters in the normal manner, or you can set them within the Common Parameters > java/javac like parameters area of the Jtest class, project, or global test parameters. This option is best if you want to override the CLASSPATH set in your system environment or if you only have a few paths to set. To point Jtest to your source files, set the Common Parameters > Source Path option in the Jtest class, project, or global test parameters. For information on setting class, project, and global test parameters, see the Jtest User’s Guide.

Black-Box Testing/Design by Contract Prerequisites

JDK Prerequisites

In order for Jtest to perform black-box (functionality) testing and use Design by Contract information, a valid path to your Java compiler must be specified in Jtest’s global test parameters.

Jtest automatically determines the path to your JDK by looking at the following variables in the order listed:

1. The PARASOFT_JDK_HOME variable.
2. The javac PATH environment variable.
3. JAVA_HOME, JDK_HOME, JAVAHOME, ...

The first valid variable found is used.

To see which JDK Jtest has detected on your system, click the Global button in the current Jtest UI, then read the value listed in the Common Parameters > Path to JDK directory branch of the Global Test Parameters window that opens.
You can configure Jtest to use a different JDK permanently or temporarily.

To permanently change the JDK used:
- Change the PARASOFT_JDK_HOME environment variable in the method appropriate for your operating system.

To temporarily change the JDK used:
1. Temporarily reset the PARASOFT_JDK_HOME variable at the command line in the method appropriate for your operating system.
2. Start Jtest from the command line as described in the Jtest User’s Guide.

**Contract Prerequisites**

If you want Jtest to automatically create black-box (functionality) test cases, your code must use Design by Contract. For information on Design by Contract, see the Jtest User’s Guide.

**JSP Testing Prerequisites**

If you want Jtest to access TLD files or “include files” referenced in a JSP, you must specify these parameters in the global, project, or class test parameters. When you are testing JSPs, you might also need to specify the URI base or URI root directories. You can specify all of these parameters in any parameter panel’s **Common Parameters** > **JSP Parameters** branch. For more details, see the Jtest User’s Guide.

**Performing a Test: Overview**

To automatically test your class(es) or JSPs with Jtest, perform the following steps:

1. Open the appropriate UI for your test. The Class Testing UI is used to test a single class or JSP; the Project Testing UI is used to test a set of classes or JSPs.
   - The Class Testing UI opens by default when Jtest is launched.
   - The Project Testing UI can be opened by clicking the Class Testing UI’s **Project** button.
2. If a class or set of classes is already loaded into the UI you are using, click the **New** button to clear the previous test.
3. Click the **Browse** button, then use the file chooser to indicate what class or set of classes you want to test.
4. Choose **Tools** > **Check Class/Project Dependencies** to determine whether Jtest can locate all required files.
5. Test the class/JSP or project for the first time by clicking the **Start** button. Jtest will open a dialog box asking you to save test parameters, then it will execute the test.

The first time you test a class or JSP, Jtest will:
- Perform static analysis (if the class’s .java source file is available or if you are testing a JSP).
- Create and execute white-box test cases that verify your code’s construction.
- Create and execute black-box test cases that verify your code’s functionality (if your code uses Design by Contract).
6. Review the class test results or project test results, then evaluate outcomes of automatically-generated test cases, correct errors found, modify the contracts, or suppress reporting of errors you do not want reported in future test runs. For details on how to do this, see the Jtest User’s Guide or Tutorial.

7. Rerun the test after you have modified one or more classes that you previously tested (i.e., perform regression testing). To do this:
   - If the class or project is not still loaded into Jtest, choose **File> Open** in the UI that you used for the original test, then choose the appropriate .ctp or .ptp file from the file chooser.
   - Click **Start**.
   When the test is run this time (and all additional times) Jtest will:
     - Repeat static analysis.
     - Repeat the previously-created test cases that are still applicable to the modified classes.
     - Create new test cases as needed for the modified code.
     - Perform regression testing by comparing the latest test case outcomes with those obtained during the initial test run (or the outcomes that you specified while evaluating test cases).

### Adding User-Defined Stubs and Test Cases (Including JUnit Test Classes)

Jtest also allows you to enter your own stubs and test cases.

Stubs can be added as Stubs Classes; for information on adding stubs, see the Jtest User’s Guide or Tutorial.

Test cases can be added as method inputs or as JUnit Test Classes. To add and execute user-defined test cases, perform these additional steps in either Jtest UI:

1. If the class or project is not still loaded into Jtest, choose **File> Open** in the UI that you used for the original test, then choose the appropriate .ctp or .ptp file from the file chooser.

2. Open the View Test Cases window to view the automatic inputs that Jtest created during previous test runs.
   - In the Class Testing UI, open the View Test Cases window by clicking **View**.
   - In the Project Testing UI, right-click the **[Class Name]** node in the Results panel, then choose **View Test Cases** from the shortcut menu.
   - **Tip:** If the Class Testing UI **View** button is not active, click the **Results** button to load results and activate the **View** button.

3. Design additional test cases as needed.

4. Add the user-defined test cases using JUnit or JTest Test Classes or method inputs as described in the following resources:
   - The Jtest User’s Guide.
   - The Jtest Tutorial.
   - “Automatically Creating a JUnit Test Class Template” on page 16.
5. Rerun the test by clicking the Start button.

When the test is run, Jtest will perform all the tests it performed in previous test runs, plus it will execute the user-defined test cases and determine the outcome for these test cases.

6. Specify the correct outcomes for any method inputs you added, as well as for automatically-generated test cases, by performing the following tasks for each class and test case:
   a. View the test case input and outcomes in the View Test Cases window.
   b. Validate correct outcomes or set the correct value for incorrect outcomes by right-clicking the appropriate outcome node, then selecting the appropriate command from the shortcut menu.

7. Rerun the test by restoring test parameters (if the class/project is not already loaded into Jtest), then clicking the Start button.

When you rerun the test, Jtest will check for specification and regression testing errors; it does this by comparing validated outcomes with their specified values, and comparing non-validated outcomes with their previous values. Jtest will also continue to test for uncaught runtime exceptions and static analysis violations.

Exploring Jtest with Sample Files

The examples in this section demonstrate how to test classes with and without Design by Contract, how to test a simple project, and how to add and execute basic user-defined test cases.

Additional examples are available in the Jtest Tutorial. The tutorial lessons cover white-box testing, black-box testing with Design by Contract, adding and executing user-defined test cases and stubs, regression testing, project testing, using Jtest with JUnit, and using Jtest in a group environment. To access the Jtest Tutorial, choose Help> Tutorial in either Jtest UI.

Testing a Simple Class

This example demonstrates how Jtest tests a single class file that does not use Design by Contract. To see how to test a sample class:

1. Go to Jtest’s Class Testing UI. (This UI opens by default when you launch Jtest).
2. If a class is already loaded into the Class Testing UI (i.e., if you see a class name in the Class Name field), click the New button to clear the previous test.
3. Browse to Simple.class (in the <jtest_install_dir>/examples/eval directory) using the Browse button in the Class Name panel.
4. Click the Start button in the tool bar, then save your class test parameters when prompted to do so.

Jtest will perform static analysis, then automatically create and execute white-box test cases designed to test the class’s construction. A dialog box will open to notify you when testing is complete. Information on test progress will be displayed in the Test Progress panel. Errors found will be reported in the Errors Found panel.

Examining Static Analysis Violations
The following coding standard violations will be reported in the Static Analysis Violations branch of the Errors Found panel.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>Avoid using text labels in &quot;switch&quot; statements. (PB.TLS-1)</td>
</tr>
<tr>
<td>[1]</td>
<td>Provide Javadoc comments for &quot;public&quot; or &quot;protected&quot; classes. (JAVADOC.PJDC-1)</td>
</tr>
<tr>
<td>[1]</td>
<td>Percentage of Javadoc comments (%). (METRICS.PJDC-2)</td>
</tr>
<tr>
<td>[1]</td>
<td>Provide a &quot;private&quot; default constructor for utility classes. (CODSTA.LCDC-2)</td>
</tr>
<tr>
<td>[2]</td>
<td>Provide Javadoc comments for &quot;public&quot; or &quot;protected&quot; methods. (JAVADOC.PJDCM-1)</td>
</tr>
<tr>
<td>[2]</td>
<td>Number of &quot;return&quot; statements. (METRICS.TRET-2)</td>
</tr>
<tr>
<td>[1]</td>
<td>Provide a file header comment for each source file. (FORMAT.MCH-2)</td>
</tr>
</tbody>
</table>

To see more information about a violation, expand the violation’s branch. For example, expand the violation of the PB.TLS rule.

This message 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. To view the source code of the class (with the line containing the violation highlighted), double-click the node containing the violation’s file/line information.

**Examining Uncaught Runtime Exceptions**

Next, let’s look at the uncaught runtime exception that Jtest’s white-box test cases uncovered. The Errors Found panel will list the following uncaught runtime exception under the Uncaught runtime exceptions branch.

This error message reveals that there is some input for which the class will throw an uncaught runtime exception at runtime. This could cause the application running this class to crash.

To see a stack trace like the one the Java virtual machine would give if this uncaught runtime exception were thrown, expand this branch.

To see an example usage of this class that would lead to the reported uncaught runtime exception, expand the Test Case Input branch.
This error message reveals that the `startsWith` method is implemented incorrectly. The method should return false for the argument "" and "0" instead of throwing a runtime exception. If the error is not fixed, any application using this class will eventually crash or give incorrect results.

To view the source code of the class (with the problematic line of the stack trace highlighted), double-click the node containing the exception's file/line information.

To see a sample of the test cases that JTest automatically created, click the View button to open the View Test Cases window. In the View Test Cases window, Control-right-click the Automatic Test Cases node, then choose Expand Children from the shortcut menu.

### Performing Regression Testing

JTest doesn't display any regression errors on the first run through a class because it is impossible to detect a regression error the first time a class is tested. Regression testing checks that class outcomes don't change, so it always needs a first run for reference.

To see how regression testing works, introduce an error into Simple.java and test it again:

1. Introduce an error into `Simple.java` as by clicking Source to open the file in an editor, then adding +3 to line 25 as follows:
2. Save the file in the editor.

3. Recompile Simple.java by right-clicking the Source button, then choosing Compile Source from the shortcut menu.

4. Click Start to retest Simple.class. Jtest then replays the existing test cases and compares the current outcomes with the previous outcomes.

The modified code changes the test outcomes, so Jtest reports the following regression errors in the Errors Found panel:

```
  ➔⇒ add: RETVAL: 3 (before was 0)
  ➔⇒ add: RETVAL: 17 (before was 14)
```
Expand the error messages to see the inputs for which these regression errors occur. The first error tells us that the method "add" is now returning 3 instead of 0 for the input 0, 0. The second error reveals that the method "add" is now returning 17 instead of 14 for the input 7,7.

Creating Reports
Jtest automatically creates a report for each test. You can view this report by clicking the Report button. By default, this report is formatted in HTML. You can configure Jtest to create reports in text (ASCII) by choosing Preferences> Configuration Options> Report Format> ASCII before you click the Report button. By default, the report file will be saved in the <jtest_install_dir>/u/username/results directory.

You can also have Jtest create an XML file of results, then apply a customized XSL file to produce a report containing the exact information and formatting you want. To generate XML-format results, right-click the Report button, then choose Create XML File from the shortcut menu. After the file is created, Jtest will open a dialog box that indicates the file’s location.

Once you have an XML result file, you can transform it using one of the XSL files shipped with Jtest or any other XSL file you want to use. Jtest’s default XSL files are located in the <jtest_install_dir>/classes/xsl directory. You can modify these XSL files so that they produce the precise type of report you need, or you can use these files as models while developing your own XSL files.

Testing a Simple Class that Uses Design by Contract
This example demonstrates how Jtest tests a sample class file that contains Design by Contract-for-format specification information. To see how to test a sample class which uses DbC:

1. Go to Jtest’s Class Testing UI. (This UI opens by default when you launch Jtest).
2. If a class is already loaded into the Class Testing UI (i.e., if you see a class name in the Class Name field), click the New button to clear the previous test.
3. Browse to Example.class (in the <jtest_install_dir>/examples/eval directory) using the Browse button in the Class Name panel.
4. Click the Start button in the tool bar, then save your class test parameters when prompted to do so.

Jtest will perform both static and dynamic analysis on the class. Because specification information is incorporated into the code using Design by Contract comment tags, Jtest can fully automate black-box (functionality) testing as well as white-box (construction) testing. Jtest will automatically create and execute black-box test cases that verify the functionality described in the class’s contracts. It will also create and execute test cases that check how the class handles a wide range of inputs.
A dialog box will open to notify you when testing is complete. Information on test progress will be displayed in the Test Progress panel. This test uncovers one Design by Contract violation, one uncaught runtime exception, and eight static analysis violations.

**Examining Design by Contract Violations**

The following Design by Contract violation will be reported in the Design by Contract Violations branch of the Errors Found panel.

This violation indicates that one of the \@post contracts was not satisfied. To see more information about this violation, expand the violation’s branch.

To open the source code of the class in an editor, click the **Source** button.

The source file reveals that the code’s \@post contract (postcondition) requires the method to return the value of \( a + b \). However, the method actually returns the value of \( a - b \). If this were your own class, you would now fix the problem (either the code or the contract), recompile the class, then retest it to verify that your modifications fixed the problem.

To see a sample of the test cases that Jtest automatically created to test this class’s functionality, click the Class Testing UI’s **View** tool bar button to open the View Test Cases window. In the View Test Cases window, Control-right-click the **Automatic Test Cases** node, then choose **Expand Children** from the shortcut menu.
Examining Uncaught Runtime Exceptions

Next, go to the uncaught runtime exception found (located in the Uncaught Runtime Exceptions branch of the Errors Found panel) and expand its branches.

This error message shows that a NegativeArraySizeException occurs when a negative index is used as an index to an array. This is an expected exception. If this were your code, you would want to document this exception in your source file by adding the following Design by Contract Java-doc comment above the method:

```
/** @exception java.lang.NegativeArraySizeException */
```

By adding this comment, you make the code easier to maintain. Someone looking at the code later on will immediately know that the method is throwing an exception because the code is supposed to throw an exception, not because the code has a bug. In addition, you configure Jtest to suppress future occurrences of this exception.

Testing a JSP
Jtest can perform static and dynamic analysis on JSP files. For an example of how to directly test a JSP:

1. Go to Jtest’s Class Testing UI. (This UI opens by default when you launch Jtest).
2. If a class is already loaded into the Class Testing UI (i.e., if you see a class name in the Class Name field), click the New button to clear the previous test.
3. Browse to `<jtest_installation_dir>/examples/jsp/num.jsp` using the Browse button in the Class Name panel.
4. Click the Start button in the tool bar, then save your class test parameters when prompted to do so. Jtest will then perform static and dynamic analysis on the `num.jsp` file. When the test is completed, Jtest will report the total number of errors found.
5. View the exceptions reported in the Errors Found Panel. Two `NumberFormatExceptions` and one `ArithmeticException` should be reported.
6. Open the branches next to one of these exceptions to view the stack trace to the exception.
7. Expand the Test Case Input branch to view the input that caused this method to throw the exception. Small white boxes to the left of a line indicate automatically-generated stubs.
8. Right-click the Test Case Input node, then choose View Example Test Case from the shortcut menu. Jtest will then display the source code of a test case that would cause this exception.
9. Click View to open the View Test Cases window where you can view the automatic test cases that Jtest created for this test.

You can also add and execute user-defined test cases for a JSP by creating a Test Class. For details on how to add user-defined test cases, see the Jtest User’s Guide.

**Automatically Creating a JUnit Test Class Template**

This example demonstrates how you can add your own test cases by modifying an automatically-created JUnit Test Class template.

You can add test cases to Jtest by specifying method inputs or using JUnit Test Classes. Test Classes let you add test cases that are too complex or difficult to be added as method inputs. Test Classes give you an easy way to write complex test cases and to verify the state of the objects under test.

You can add test cases via Test Classes in any of the following ways:

- Have Jtest create a JUnit-compatible Test Class template, then add test cases by modifying the template.
- Have Jtest export user-defined or automatically-generated inputs as JUnit-compatible Test Classes, then add more test cases by modifying the Test Classes.
- Write your own Test Classes using any Java development environment.
- Integrate JUnit classes into your Jtest tests.

You can run any automatically-generated JUnit-compatible Test Class or Test Class template in Jtest or JUnit.

For an example of how to create a JUnit Test Class Template for a sample class file, add a user-defined test case to that Test Class, and run the test case in Jtest:

1. Go to Jtest’s Class Testing UI. (This UI opens by default when you launch Jtest).
2. If a class is already loaded into the Class Testing UI (i.e., if you see a class name in the Class Name field), click the New button to clear the previous test.

3. Browse to Simple.class (in the <jtest_install_dir>/examples/eval directory) using the Browse button in the Class Name panel.

4. Click Class to open the Class Test Parameters window.

5. Right-click the Class Test Parameters tree’s Dynamic Analysis> Test Case Generation> User Defined> Test Classes node, then choose Add JUnit Test Class Template from the shortcut menu. A file chooser will open so you can enter a name and path. For this example, save this file as SimpleTest.java in the eval directory.

   After you select a name and location for the template, Jtest will open the generated template in the designated editor. The Test Class contains a test for each of the methods in the class under test. By default, each test will fail until you add your own test code. The Test Class also includes comments that describe the purpose of each method, which method the test method is testing, and where you need to modify or add to the code.

6. Add the following code to the test method testAdd() to create a valid test case for the Add method which adds 5 and 88, then verifies whether the correct response (93) is achieved:

   ```java
   public void testAdd ()
   {
       int result = 0;
       result = Simple.add (5, 88);
       assertTrue ("The result should be 93", result == 93);
   }
   ```

7. Remove the comments that were originally included for this method.

8. Save your modifications.

9. Compile the Test Class as follows:
   a. Click Class to open the Class Test Parameters window.
   b. Expand the Class Test Parameters tree’s Dynamic Analysis> Test Case Generation> User Defined> Test Cases branch, then right-click the node representing the Test Class (examples.eval.SimpleTest) and choose Compile Test Class from the shortcut menu.

   **Important:** Make sure that junit.jar is in your CLASSPATH so you will be able to compile the Test Class.

You can now use the Test Class within Jtest or JUnit. To run the Test Class in Jtest, click Start. Each test method that you have not yet modified will fail because the template contains fail statements for each test method. Jtest reports an error message for each failure to remind you to modify the test class template.

**Note:** The junit.jar file needs to be on the CLASSPATH before you use the Test Class in either Jtest or JUnit.

### Exporting Automatically-Generated Test Cases as JUnit-Compatible Test Classes

This example demonstrates how to export automatically-generated test cases as a JUnit-compatible Test Class that can be run in JUnit or Jtest.
If you want to run any or all of Jtest's automatically-generated test cases in the JUnit framework, you can use the Export to a JUnit Test Class feature to export the test cases as a JUnit-compatible Test Class, then use that Test Class in JUnit without having to make any modifications.

If you want to simplify test case creation, you can use this feature to create a Test Class that represents existing test cases, duplicate and modify the existing test cases as needed, then run the modified Test Class in Jtest or JUnit.

For an example of how to create a Test Class based on automatically-generated inputs:

1. Go to Jtest's Class Testing UI. (This UI opens by default when you launch Jtest).
2. If a class is already loaded into the Class Testing UI (i.e., if you see a class name in the Class Name field), click the New button to clear the previous test.
3. Browse to Simple.class (in <jtest_install_dir>/examples/eval) using the Browse button in the Class Name panel.
4. Verify whether Jtest can locate all necessary files by choosing Tools> Check Class Dependencies. If Jtest cannot locate a necessary file, it will open a dialog box that allows you to modify your CLASSPATH to point to the necessary file.
5. Click Start to start the test.
6. To see a selection of the test cases that Jtest automatically generated for this test, click the View button in the tool bar after the Simple.class test is completed. The View Test Cases window will open.
7. Export the automatic test cases to a JUnit Test Class by right-clicking the Automatic Test Cases branch, then choosing Export to a JUnit Test Class from the shortcut menu. Save this Test Class as SimpleTestAuto.java in the eval directory.

After you select a name and location for the Test Class, it will open in the designated editor so you can modify it if desired. The Test Class created tests each method with the arguments created during dynamic analysis. The return value is then compared to the results from dynamic analysis using the JUnit assertTrue methods. Note that Jtest created 14 test methods that correspond to the generated test cases.

8. Make any necessary changes to the methods. The test methods, the setUp method, and tearDown method can be modified to your specifications.
9. Save your modifications.
10. Compile the Test Class as follows:
   a. Click Class to open the Class Test Parameters window.
   b. Expand the Class Test Parameters tree’s Dynamic Analysis> Test Case Generation> User Defined> Test Cases branch, then right-click the node representing the Test Class and choose Compile Test Class from the shortcut menu.

Important: Make sure that junit.jar is in your CLASSPATH so you will be able to compile the Test Class.

You can now use the Test Class within Jtest or JUnit. To run the Test Class in Jtest, click Start.

Note: The junit.jar file needs to be on the CLASSPATH before you use the Test Class in either Jtest or JUnit.

Adding Test Cases as Method Inputs
This example demonstrates how to add primitive input values to a static method and how to validate the outcomes for these inputs.

To add and validate the sample method inputs:

1. Go to Jtest's Class Testing UI. (This UI opens by default when you launch Jtest).

2. If a class is already loaded into the Class Testing UI (i.e., if you see a class name in the **Class Name** field), click the **New** button to clear the previous test.

3. **Browse** to StaticInstanceExample.class (in the `<jtest_install_dir>/examples/tutorial/lesson3` directory) using the **Browse** button in the Class Name panel.

4. Click the **Class** button to open the Class Test Parameters window.

5. Open the Class Test Parameters' **Dynamic Analysis**> **Test Case Generation**> **User Defined**> **Method Inputs**> **returnStaticInt** node.

6. Right-click **int ARG1**, then choose **Add Input Value** from the shortcut menu.

7. Type **3** in the text field, then press **Enter**.

8. To check the test case that you just added, press the `<control>` key and right-click the **User Defined** node, then select **Check** from the shortcut menu.

9. Verify whether Jtest can locate all necessary files by choosing **Tools**> **Check Class Dependencies**. If Jtest cannot locate a necessary file, it will open a dialog box that allows you to modify your CLASSPATH to point to the necessary file.

10. Run the test by closing the Class Test Parameters window, then clicking **Start** in the Class Testing UI.

11. When the test is complete, view the results of the user-defined test case, then validate the outcome.

    a. Click **View** to open the View Test Cases window.
Testing a Project

This example demonstrates how to test a set of files. In Jtest's Project Testing UI, you can automatically test all (or a selected set) of the classes and JSPs contained in any directory, jar file, or zip file. Jtest automatically searches the specified directory, jar file, or zip file, and tests all of the classes and JSPs that it finds.

To perform automatic testing (white-box testing and static analysis) on all files in a sample directory:

1. Click the **Project** button in the Class Testing UI tool bar. The Project Testing UI will open.

2. Click the **Browse** button in the Project Controls panel, select the sample Jtest project directory (`<jtest_install_dir>/examples/eval/project`) in the file chooser that opens, then click **Open**.

3. Click the **Start** button in the Project UI tool bar.

Jtest will prompt you to save your test parameters, then it will start finding and testing classes.

Errors found will be reported in the **[Class Name]- Errors Found** branch of the Project Testing UI's Results panel. Test results for each class are available in the lower results panel. These results are organized by class name. The structure of results for each class is similar to the structure of results in the Class Testing UI's Errors Found panel.

To produce an HTML-format report of these results, click **Report**.

b. Open **User Defined Test Cases> Method Inputs> returnStaticInt> Test Case 1**.

c. Right-click **RETVAL=3**, then choose **Mark as Correct** to indicate that this is the correct outcome for the given input.
To make the lower results panel display only a certain type of result (such as Uncaught Runtime Exceptions or java.lang.NullPointerExceptions) perform the following action:

- Right-click the Number of Errors Found panel node that describes the type of results you want to view, then choose **Show Results for This Category** from the shortcut menu.