User's Guide

Version 4.1

ParaSoft Corporation
2031 S. Myrtle Ave.
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com
URL: www.parasoft.com

PARASOFT END USER LICENSE AGREEMENT

REDISTRIBUTION NOT PERMITTED

This Agreement has 3 parts. Part I applies if you have not purchased a license to the accompanying software (the "SOFTWARE"). Part II applies if you have purchased a license to the SOFTWARE. Part III applies to all license grants. If you initially acquired a copy of the SOFTWARE without purchasing a license and you wish to purchase a license, contact ParaSoft Corporation ("PARASOFT"):

(626) 305-0041

(888) 305-0041 (Toll-Free)

(626) 305-9048 (Fax)

info@parasoft.com

http://www.parasoft.com

PART I -- TERMS APPLICABLE WHEN LICENSE FEES NOT (YET) PAID GRANT.

DISCLAIMER OF WARRANTY.

Free of charge SOFTWARE is provided on an "AS IS" basis, without warranty of any kind, including without limitation the warranties of merchantability, fitness for a particular purpose and non-infringement. The entire risk as to the quality and performance of the SOFTWARE is borne by you. Should the SOFTWARE prove defective, you and not PARASOFT assume the entire cost of any service and repair. This disclaimer of warranty constitutes an essential part of the agreement. SOME JURISDICTIONS DO NOT ALLOW EXCLUSIONS OF AN IMPLIED WARRANTY, SO THIS DISCLAIMER MAY NOT APPLY TO YOU AND YOU MAY HAVE OTHER LEGAL RIGHTS THAT VARY BY JURISDICTION.

PART II -- TERMS APPLICABLE WHEN LICENSE FEES PAID

GRANT OF LICENSE.

PARASOFT hereby grants you, and you accept, a limited license to use the enclosed electronic media, user manuals, and any related materials (collectively called the SOFTWARE in this AGREEMENT). You may install the SOFTWARE in only one location on a single disk or in one location on the temporary or permanent replacement of this disk. If you wish to install the SOFTWARE in multiple locations, you must either license an additional copy of the SOFTWARE from PARASOFT or request a multi-user license from PARASOFT. You may not transfer or sub-license, either temporarily or permanently, your right to use the SOFTWARE under this AGREEMENT without the prior written consent of PARASOFT.

LIMITED WARRANTY.

PARASOFT warrants for a period of thirty (30) days from the date of purchase, that under normal use, the material of the electronic media will not prove defective. If, during the thirty (30) day period, the software media shall prove defective, you may return them to PARASOFT for a replacement without charge.

THIS IS A LIMITED WARRANTY AND IT IS THE ONLY WARRANTYMADE BY PARASOFT. PARASOFT MAKES NO OTHER EXPRESS WARRANTY AND NO WARRANTY OF NONINFRINGEMENT OF THIRD PARTIES' RIGHTS. THE DURATION OF IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION, WARRANTIES OF MERCHANTABILITY AND OF FITNESS FOR A PARTICULAR PURPOSE, IS LIMITED TO THE ABOVE LIMITED WARRANTY PERIOD; SOME JURISDICTIONS DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO LIMITATIONS MAY NOT APPLY TO YOU. NO PARASOFT DEALER, AGENT, OR EMPLOYEE IS AUTHORIZED TO MAKE ANY MODIFICATIONS, EXTENSIONS, OR ADDITIONS TO THIS WARRANTY.

If any modifications are made to the SOFTWARE by you during the warranty period; if the media is subjected to accident, abuse, or improper use; or if you violate the terms of this Agreement, then this warranty shall immediately be terminated. This warranty shall not apply if the SOFTWARE is used on or in conjunction with hardware or software other than the unmodified version of hardware and software with which the SOFTWARE was designed to be used as described in the Documentation. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY HAVE OTHER LEGAL RIGHTS THAT VARY BY JURISDICTION.

YOUR ORIGINAL ELECTRONIC MEDIA/ARCHIVAL COPIES.

The electronic media enclosed contain an original PARASOFT label. Use the original electronic media to make "back-up" or "archival" copies for the purpose of running the SOFTWARE program. You should not use the original electronic media in your terminal except to create the archival copy. After recording the archival copies, place the original electronic media in a safe place. Other than these archival copies, you agree that no other copies of the SOFTWARE will be made.

TERM.

This AGREEMENT is effective from the day you install the SOFTWARE and continues until you return the original SOFTWARE to PARASOFT, in which case you must also certify in writing that you have destroyed any archival copies you may have recorded on any memory system or magnetic, electronic, or optical media and likewise any copies of the written materials.

CUSTOMER REGISTRATION.

PARASOFT may from time to time revise or update the SOFTWARE. These revisions will be made generally available at PARASOFT's discretion. Revisions or

notification of revisions can only be provided to you if you have registered with a PARASOFT representative or on the ParaSoft Web site. PARASOFT's customer services are available only to registered users.

PART III -- TERMS APPLICABLE TO ALL LICENSE GRANTS

SCOPE OF GRANT.

DERIVED PRODUCTS.

Products developed from the use of the SOFTWARE remain your property. No royalty fees or runtime licenses are required on said products.

PARASOFT'S RIGHTS.

You acknowledge that the SOFTWARE is the sole and exclusive property of PARASOFT. By accepting this agreement you do not become the owner of the SOFTWARE, but you do have the right to use the SOFTWARE in accordance with this AGREEMENT. You agree to use your best efforts and all reasonable steps to protect the SOFTWARE from use, reproduction, or distribution, except as authorized by this AGREEMENT. You agree not to disassemble, de-compile or otherwise reverse engineer the SOFTWARE.

SUITABILITY.

PARASOFT has worked hard to make this a quality product, however PARASOFT makes no warranties as to the suitability, accuracy, or operational characteristics of this SOFTWARE. The SOFTWARE is sold on an "as-is" basis.

EXCLUSIONS.

PARASOFT shall have no obligation to support SOFTWARE that is not the then current release.

TERMINATION OF AGREEMENT.

If any of the terms and conditions of this AGREEMENT are broken, this AGREEMENT will terminate automatically. Upon termination, you must return the software to PARASOFT or destroy all copies of the SOFTWARE and Documentation. At that time you must also certify, in writing, that you have not retained any copies of the SOFTWARE.

LIMITATION OF LIABILITY.

You agree that PARASOFT's liability for any damages to you or to any other party shall not exceed the license fee paid for the SOFTWARE.

PARASOFT WILL NOT BE RESPONSIBLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OF THE SOFTWARE ARISING OUT OF ANY BREACH OF THE WARRANTY, EVEN IF PARASOFT HAS BEEN ADVISED OF SUCH DAMAGES. THIS PRODUCT IS SOLD "AS-IS".

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

ENTIRE AGREEMENT.

This Agreement represents the complete agreement concerning this license and may be amended only by a writing executed by both parties. THE ACCEPTANCE OF ANY PURCHASE ORDER PLACED BY YOU IS EXPRESSLY MADE CONDITIONAL ON YOUR ASSENT TO THE TERMS SET FORTH HEREIN, AND NOT THOSE IN YOUR PURCHASE ORDER. If any provision of this Agreement is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This Agreement shall be governed by California law (except for conflict of law provisions).

ParaSoft Corporation

2031 South Myrtle Avenue

Monrovia, CA 91016

Printed in the U.S.A, July 30, 2001

Jtest User's Guide Table of Contents

Introduction

Testing With Jtest

Static Analysis

Dynamic Analysis

White-Box Testing

Black-Box Testing

Design by Contract

Regression Testing

IDE Integration

Test-Related Tasks

Customizing Your Test

Jtest UI Help

Class Testing UI

Project Testing UI

Test Parameters Windows

Tools

Reference

Index

Introduction

Welcome to Jtest, a Java unit testing tool that automatically tests any Java class or component without requiring you to write a single test case, harness, or stub.

The development community endorses practices such as unit testing, coding standard enforcement, and Design by Contract. When implemented, these techniques prevent software errors, increase code stability, and automate the testing techniques that are a fundamental part of any Extreme Programming process. Until now, these practices were too labor-intensive and costly to implement. Jtest fully automates these practices, so even the most time-stricken developers can fit these procedures into their schedules.

Jtest automates black-box (functionality) testing, white-box (construction) testing, and regression testing. In addition, it automatically enforces over 240 industry-respected coding standards, lets you tailor standards for a specific project or group, and enables you to create and enforce customized coding standards.

By testing at the unit, or class, level, Jtest 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.

Windows Installation and Setup

Requirements

- Windows NT Service Pack 6 or Windows 2000
- JDK 1.2 or higher

Installation

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.

Startup

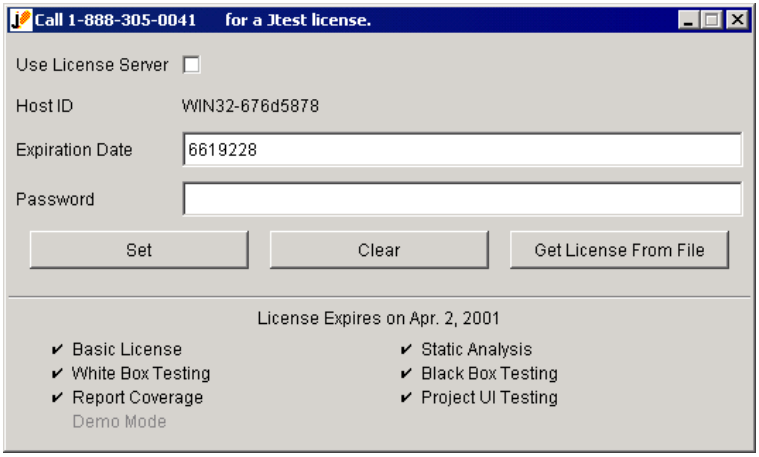
To launch Jtest, double-click the Jtest desktop icon.

A Jtest license must be installed before you can begin using Jtest.

Installing a License

To install a machine-locked Jtest license on your machine:

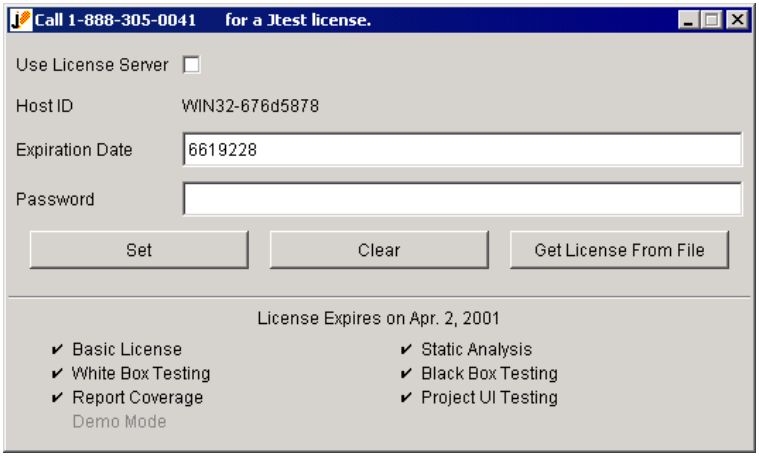
1. Launch Jtest as described above. The Class Testing UI and the License window will open.



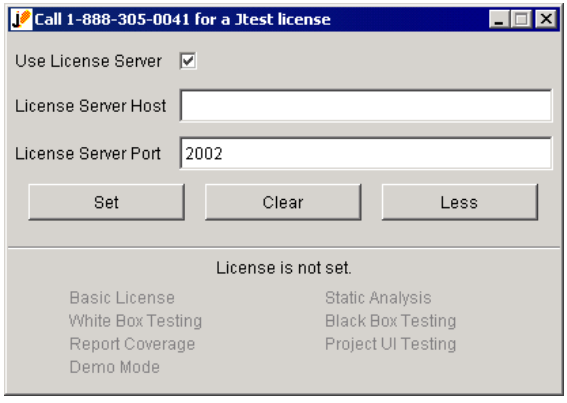
2. Call 1-888-305-0041 to get your license.
3. In the License window, enter your expiration date and password.
4. Click **Set** to set and save your license.

To install a network license and have ParaSoft's LicenseServer manage license access across your local area network:

1. Launch Jtest as described above. The Class Testing UI and the License window will open.



2. In the License window, check the **Use License Server** option. The License window will then change.



3. Enter your LicenseServer host in the **License Server Host** field.

4. Enter your LicenseServer port in the **License Server Port** field (the default port is 2002).
5. Click **Set** to set and save your LicenseServer information.
6. Call 1-888-305-0041 to get your license.
7. Add your license to the LicenseServer as described in the LicenseServer documentation.

UNIX Installation and Setup

Glossary

<jtest-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 6.2 or 7.1 with one of the following kernels: 2.2.14-5.0, 2.4.2-2.

Installing Jtest

1. Copy the jtest.<arch>.tar.<compression-scheme> to the directory where you would like to install Jtest.
2. Extract the archive. During extraction, a directory named 'jtest' will be created with the program files necessary to run the program.
    - 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>.

Setting the Environment

After installing Jtest, you must set up your environment before you can run Jtest. To set the environment:

1. Use the provided shell script to set up your environment or set up the environment by hand.
    - To use the script:
        - For bash or sh shells: Run the 'jtvars.sh' script in <jtest-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 up the environment by hand:
      The script sets up a couple of environment variables needed to run Jtest. It adds to the PATH environment variable the '<jtest-home>/bin' directory. Additionally, it adds to the LD_LIBRARY_PATH environment variable the '<jtest-home>/lib' directory.
2. Add 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 Sun Microsystem's JDK on your machine.
    - bash or sh shell example:
      ```
      $ PARASOFT_JDK_HOME=/usr/java/jdk1.3.1
      $ export PARASOFT_JDK_HOME
      ```

- tcsh, csh or ksh shell example:
  `$ setenv PARASOFT_JDK_HOME=/usr/java/jdk1.3.1`
- **Note:** If you add the 'bin' directory of the JDK from Sun to your environment, 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" from Sun on your PATH.
   If you are confused about this step, then it is best to ask a sysadmin for help. Until the sysadmin responds, use the scripts provided in the <jtest-home> directory.

Starting Jtest

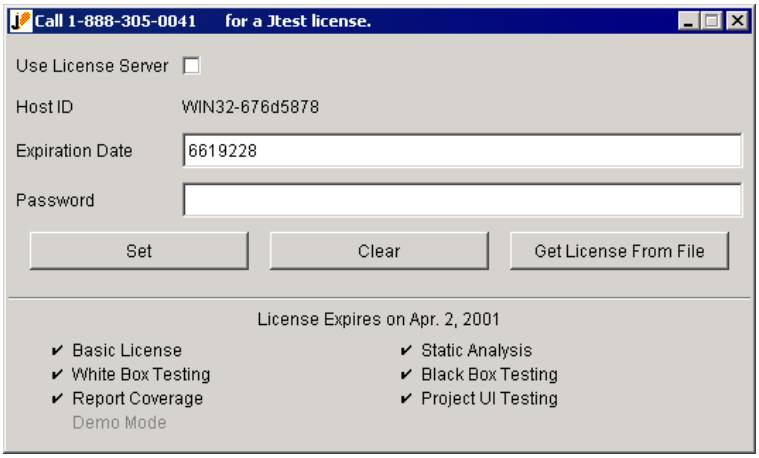
Once the environment has been set, Jtest is started by running the `jtest-gui` command.

A Jtest license must be installed before you can begin using Jtest.

Installing a License

To install a machine-locked Jtest license on your machine:

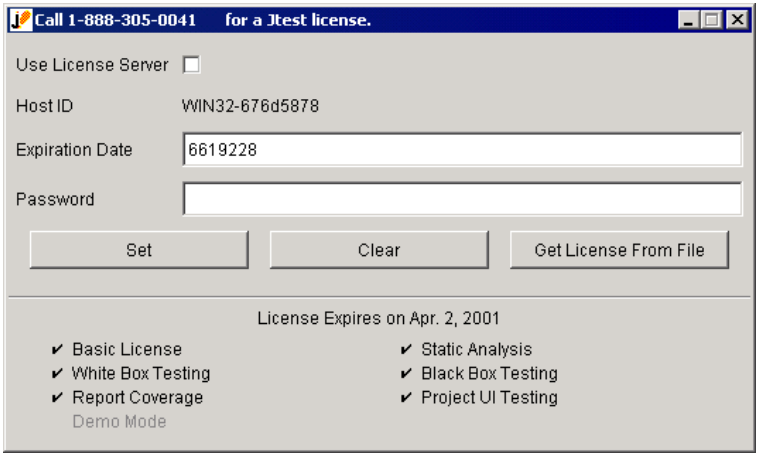
1. Launch Jtest as described above. The Class Testing UI and the License window will open.



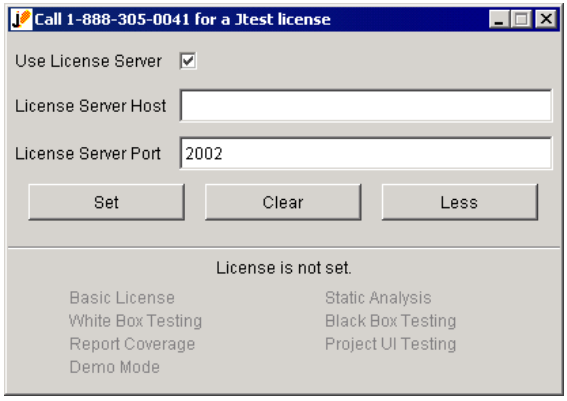
2. Call 1-888-305-0041 to get your license.
3. In the License window, enter your expiration date and password.
4. Click **Set** to set and save your license.

To install a network license and have ParaSoft's LicenseServer manage license access across your local area network:

1. Launch Jtest as described above. The Class Testing UI and the License window will open.



2. In the License window, check the **Use License Server** option. The License window will then change.



3. Enter your LicenseServer host in the **License Server Host** field.

4. Enter your LicenseServer port in the **License Server Port** field (the default port is 2002).
5. Click **Set** to set and save your LicenseServer information.
6. Call **1-888-305-0041** to get your license.
7. Add your license to the LicenseServer as described in the LicenseServer documentation.

If Jtest Cannot Locate Your JDK

If Jtest ever 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 'bin' directory of the JDK 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
    ```

Contacting ParaSoft

ParaSoft is committed to providing you with the best possible product support for Jtest. If you have any trouble installing or using Jtest, please follow the procedure below in contacting our Quality Consulting department.

- Check the manual.
- Be prepared to recreate your problem.
- Know your Jtest version. (You can find it in **Help> About**.)

Jtest experts are available online to answer your questions. To receive live online support, choose **Help> Support** to open the Jtest support page, then follow the link to “Live Online Jtest Experts.”

Contact Information

- **USA Headquarters**

  Tel: (888) 305-0041

  Fax: (626) 305-9048

  Email: jtest@parasoft.com

  Web Site: http://www.parasoft.com

- **ParaSoft France**

  Tel: +33 (0) 1 64 89 26 00

  Fax: +33 (0) 1 64 89 26 10

  Email: jtest@parasoft-fr.com

- **ParaSoft Germany**

  Tel: +49 (0) 78 05 95 69 60

  Fax: +49 (0) 78 05 95 69 19

  Email: quality@parasoft-de.com

- **ParaSoft UK**

  Tel: +44 (020) 8263 2827

  Fax: +44 (020) 8263 2701

  Email: quality@parasoft-uk.com

Quick Start Guide

Jtest fully automates white-box testing, black-box testing, regression testing, and static analysis. The only user intervention required to perform these tests on a class or set of classes is telling Jtest what class or set of classes to test, clicking the **Start** button, and looking at the test results.

Requirements

General Requirements

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

- The '.class' files for the classes you want to test must be available. A '.class' file is a compiled Java source. Without a '.class' 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 referenced by the tested '.class' files must be available to Jtest. This is done by adding their location to the CLASSPATH.
- If the '.class' files are in directories, '.zip' files, or '.jar' files, the '.class' files must be accessible by Jtest.

To use full Jtest functionality (Static Analysis, Source Browsing, Design by Contract, etc.) the '.java' source files must be available to Jtest during testing.

Black-Box Testing/Design by Contract Requirements

JDK Requirement

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 present 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 what 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 change the JDK permanently:

- Change the PARASOFT_JDK_HOME environment variable in the method appropriate for your operating system.

To change the JDK temporarily:

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 "Running Jtest in Batch Mode" on page 60.

Contract Requirement

In order for Jtest to automate black-box testing, code must contain Design by Contract-format contracts. For information on adding Design by Contract-format contracts to your code, see "Using Design by Contract With Jtest" on page 133 and "The Design by Contract Specification Language" on page 141.

General Testing Procedure

To test your class(es) 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; the Project Testing UI is used to test a set of classes.
    - 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. Use the **Browse** button to indicate what class or set of classes you want to test.
4. Test the class or project for the first time by clicking the **Start** button.

    The first time you test a class, Jtest will:
    - Perform static analysis (if the class's .java source file is available).
    - Create and execute white-box test cases that check your code's construction.
    - Create and execute black-box test cases that verify your code's functionality (if your code contains Design by Contract-format contracts).
5. Review the class test results or project test results, then correct errors found, modify the contracts, or suppress reporting of errors you do not want reported in future test runs.
6. Rerun the test when a class is modified (i.e., perform regression testing). To do this:
    a. 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.
    b. Click **Start**.

When the test is run this time (and all additional times) Jtest will:

- Perform static analysis.
- Create and execute white-box test cases that check your class's construction.
- Create and execute black-box test cases that verify your class's functionality (if your code contains Design by Contract-format contracts).
- Perform regression testing by comparing the current test case outcomes with those obtained during the initial test run.

Adding User-Defined Stubs and Test Cases

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

Stubs can be added as Stub Classes; for information on adding stubs, see "Using Custom Stubs" on page 98.

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

1. Open the View Test Cases window to view the automatic inputs that Jtest generated 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.
2. Design additional test cases.
3. Add the user-defined test cases using Test Classes or method inputs.
    - For information on adding Test Classes, see "Adding Test Classes" on page 125.

- For information on adding method inputs, see “Adding Method Inputs” on page 119.

4. 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 output for the user-defined test cases.

5. Specify the correct outcomes for the user-defined test cases, 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 and selecting the appropriate command from the shortcut menu.
   c. (optional) Specify additional inputs to check.

6. When the class is modified, rerun the test by restoring test parameters and 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 nonvalidated outcomes with their previous values. Jtest will also continue to test for uncaught runtime exceptions and static analysis errors.

Setting Your CLASSPATH

If during testing, Jtest finds ClassNotFoundExceptions or NoClassDefFoundErrors, or if it reports that it could not find the package on "imports", the CLASSPATH is not set properly. If this occurs, you need to set the system CLASSPATH variable to include every class referenced (recursively) by the tested class prior to testing. Check that the CLASSPATH includes the parent directory of the directory hierarchy. For example, if you are testing com.company.MyClass and Jtest reports that it could not find a package referenced by MyClass, it is probably because the 'com' directory is not on the CLASSPATH.

You can override the CLASSPATH environment variable in the Global Test Parameters, the Class Test Parameters, or the Project Test Parameters.

Testing a Single Class

To test a single class:

1. Indicate what class to test by performing one of the following steps:
    - Browse for the class by...
        a. Clicking the **Browse** button in the Class Testing UI's Class Name panel.
        b. Locating and selecting the .class file you want to test in the file viewer.
        c. Clicking **Open**.
    - Enter the fully qualified name of the class to test (without the .class extension) in the **Class Name** field.

      Note: We recommended that you use the Class Testing UI's **Browse** button to select the class you want to test. When a class is selected using the **Browse** button, the working directory is set to the root directory of the class's package.
    - Use the Find Classes UI to find available classes, then double-click the name of the appropriate test in the lower panel of the Find Classes UI. This will set up a test for the selected class in the Class Testing UI.
2. Start the test by clicking the **Start** tool bar button.
    - If you only want to perform static analysis or a specific type of static analysis, right-click the **Start** button and choose the menu item that describes the type of test that you want to perform.
    - If you only want to perform dynamic analysis or a specific type of dynamic analysis, right-click the **Start** button and choose the menu item that describes the type of test that you want to perform.

Unless you tell it to do otherwise, Jtest automatically performs all steps required for:

- Static analysis
- White-box testing
- Black-box testing (if Design by Contract-format contract information is included in the class under test or test case outcomes have been validated)
- Regression testing (on all test runs after the first)

If you want to configure Jtest to perform only static analysis or only dynamic analysis, modify your testing parameters as described in “Customizing Test Parameters” on page 180.

For details on specific types of tests performed, see the following topics:

- “About Static Analysis” on page 69
- “About Dynamic Analysis” on page 85
- “About White-Box Testing” on page 108
- “About Black-Box Testing” on page 113
- “About Regression Testing” on page 152

Results

Results are displayed in the Errors Found Panel. To learn more about this panel's branches and available options, see “Understanding the Errors Found Panel” on page 32 and “Exploring and Customizing Class Test Results” on page 37.

Testing A Class - Two Simple Examples

The following examples demonstrates how to perform fully automatic testing on two classes: one without Design by Contract comments and one with these comments.

Example 1: Testing a Class Without Design by Contract Comments

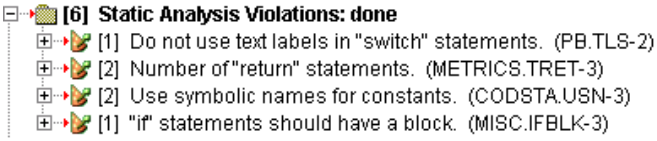
This example demonstrates how Jtest tests a single class file that does not contain Design by Contract comments.

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. Click the **Start** button in the tool bar.

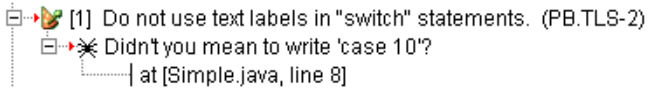
Jtest will perform static analysis, then automatically create and execute white-box test cases designed to test the class's construction. A dialog box will open to notify you when testing is complete. Information on test progress will be displayed in the Test Progress panel. Errors found will be reported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

Static Analysis Violations

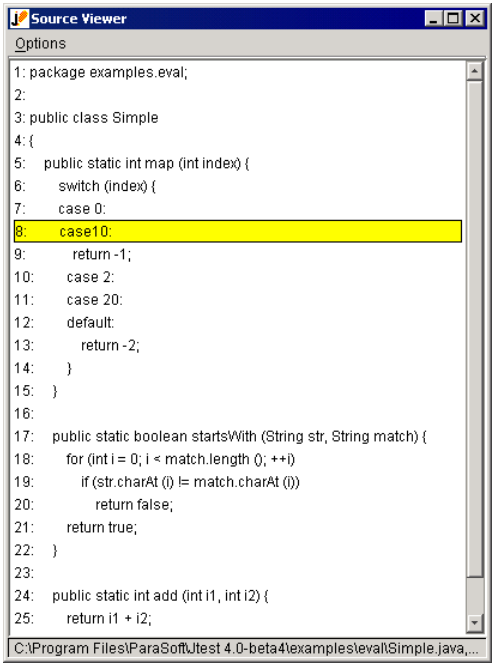
The following coding standard violations will be reported in the **Static Analysis Violations** branch of the Errors Found panel.



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

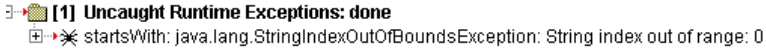


This violation reveals that the developer inadvertently wrote `case10` instead of `case 10`. If the class is not fixed, it will give incorrect results when it is passed the value `10`. To view the source code of the class (with the line containing the error highlighted), double-click the node containing the error's file/line information.



Uncaught Runtime Exceptions

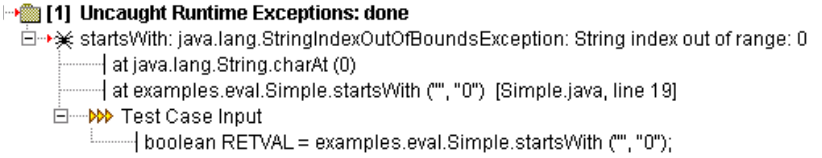
Now let's look at the uncaught runtime exception that Jtest's white-box test cases found. The Errors Found panel will list the following uncaught runtime exception under the **Uncaught Runtime Exceptions** branch.



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.

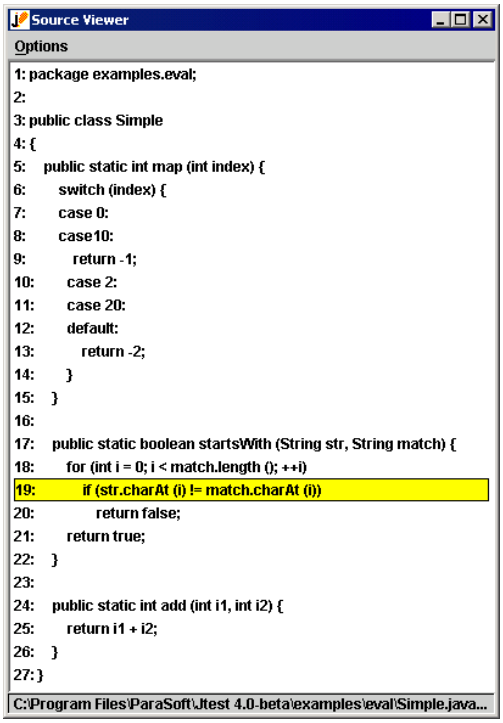
To see a stack trace like the one the Java virtual machine would give if this uncaught runtime exception were thrown, expand this branch.

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.

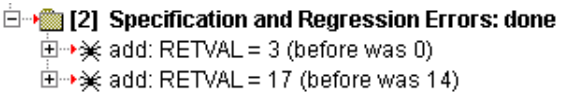
Regression Testing

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.

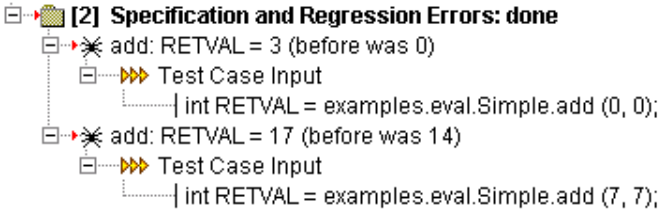
To see how regression testing works, introduce an error into Simple.java and run it again.

1. Copy Simple_re.java into Simple.java (both classes are located in <jtest_install_dir>/examples/eval).
2. Recompile Simple.java.
3. Retest Simple.class.

Now, along with the other errors, Jtest reports the following regression errors in the Errors Found panel:



Expand the error messages to see the inputs for which these regression errors occur. The first error tells us that the code has changed and 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.



Example 2: Testing a Class With Design by Contract Comments

This example demonstrates how Jtest tests a single class file that contains Design by Contract-format specification information.

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 <jtest_install_dir>/examples/eval) using the **Browse** button in the Class Name panel.
4. Click the **Start** button in the tool bar.

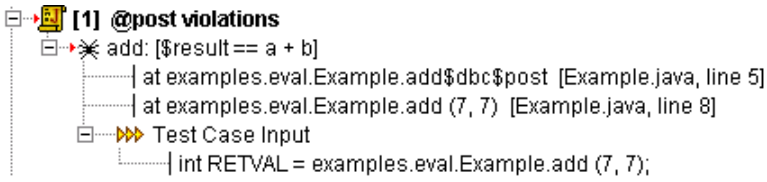
Jtest will perform both static and dynamic on the class. Because the code's specification is incorporated into the code using Design by Contract comment tags, Jtest can fully automate black-box (functionality) testing as well as white-box (construction) testing. Jtest will automatically create and execute black-box test cases that verify the functionality described in the class's postcondition contract. It will also create and execute white-box 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 one static analysis violation.

Design by Contract Violations

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

This violation indicates that one of the class's methods did not function as described in the specification. To see more information about this violation, expand the violation's branch.



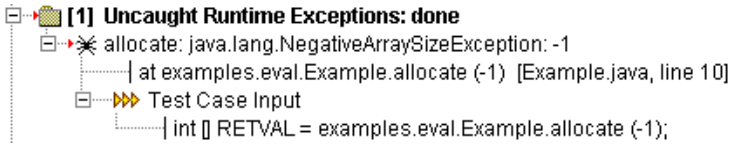
To open the source code of the class in an editor, right-click the **Source** button, then choose **Edit Source** from the shortcut menu.

The source file reveals that the code's `@post` contract (postcondition) requires the method to return the value of `a+b`. However, as Jtest revealed, it does not function as specified: the method actually returns the value of `a-b`. If this were your own class, you would now fix the problem, recompile the class, then retest it to check if 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 **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.

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 code by adding the following Design by Contract Javadoc comment above the method:

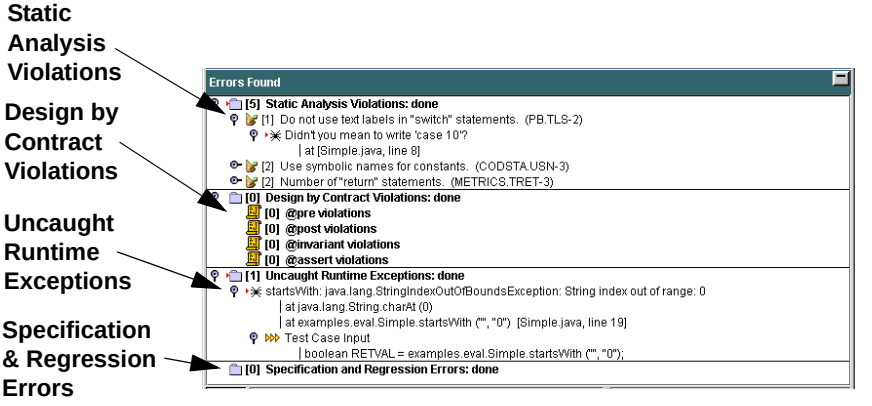
```
/** @exception java.lang.NegativeArraySizeException */
```

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 tell Jtest to suppress future occurrences of this exception.

Understanding the Errors Found Panel

All class test results are displayed in the Class Testing UI's Errors Found panel. The contents of this panel are described below, and in context-sensitive help. If commands are available by right-clicking a particular node, a right-click icon will open when you place your cursor over that node.

The Class Testing UI also provides you with a variety of ways to gain additional information about the test results and to customize what results are reported the next time that this test is run. For more information about these options, see "Exploring and Customizing Class Test Results" on page 37.

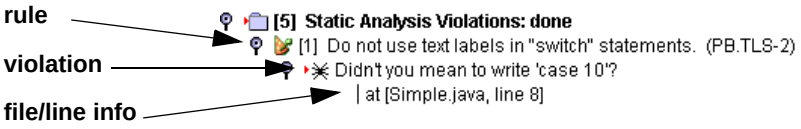


All error/violation messages are marked with a bug icon.

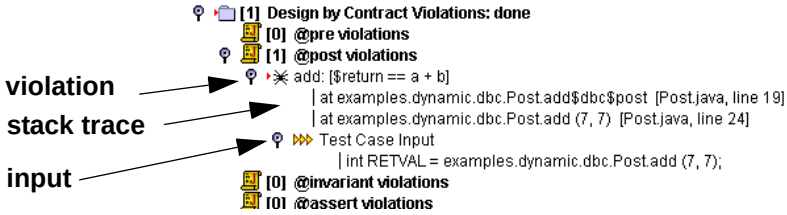
The tree displayed in this panel contains the following information:

- **Static Analysis Violations:** Displays the number of violations that Jtest found while performing static analysis. This branch contains the following information:

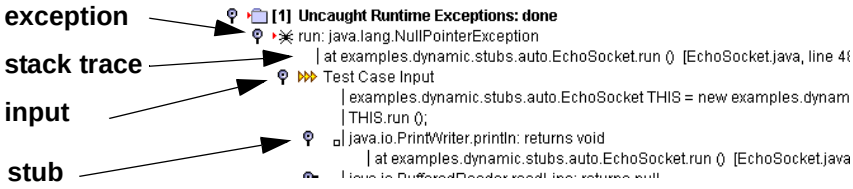
- **Rule:** Name of rule violated. (Rule ID is displayed in parentheses).
  Marked with a wizard hat icon.
- **Violation:** Jtest rule violation message. To suppress this message or view the associated rule description, right-click this node then choose the appropriate command from the shortcut menu.
  Marked with a bug icon.
- **File/line info:** File/line number where violation occurred. To view or edit the source code, right-click this node then choose the appropriate command from the shortcut menu.



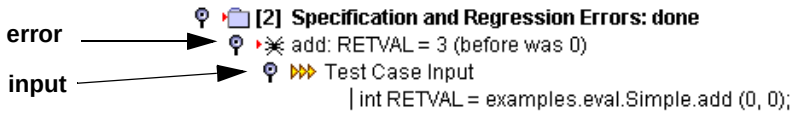
- **Design by Contract Violations:** Displays the number of Design by Contract violations that Jtest found while performing dynamic analysis. Design by Contract violations are organized according to the nature of the violation. This branch contains the following violation categories:
  - **@pre violations:** Contains information about violations that occur when a method is called incorrectly.
  - **@post violations:** Contains information about violations that occur when a method does not return the expected value.
  - **@invariant violations:** Contains information about violations that occur when an @invariant contact condition is not met.
  - **@assert violations:** Contains information about violations that occur when an @assert contact condition is not met.



- **Uncaught Runtime Exceptions:** Displays the number of uncaught runtime exceptions that Jtest found while performing dynamic analysis. Each uncaught runtime exception is followed by a full stack trace, as well as an example input leading to this exception. This branch contains the following information:
  - **Exception:** Exception found.
    Marked with a bug icon.
  - **Stack trace information:** A stack trace like the one that the Java virtual machine would give if a reported uncaught runtime exception were thrown. To view or edit the source code, right-click this node then choose the appropriate command from the shortcut menu. (If the file and line number information is missing, recompile the class with debug information).
  - **Input that defines the test case:** For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument.
    If input from a stub caused the exception, stub information will be displayed here. Empty boxes indicate automatically generated stubs. Black boxes indicate user-defined stubs. To see the stack trace where a stub invocation occurred, expand the stub's branch. For more information on stubs, see "Testing Classes That Reference External Resources" on page 93 and "Using Custom Stubs" on page 98.
    Marked with an arrow icon.



- **Specification and Regression Errors:** Displays the specification and regression errors that Jtest found while performing dynamic analysis. The errors are determined by comparing the test case outcomes of this run with those of previous runs or those specified by the user. (To view the reference outcomes, open the Class Test Parameters window and browse to **Dynamic Analysis> Test Case Evaluation> Specification and Regression Test Cases**). This branch contains the following information:
  - **Error:** Specification and regression error found.
    Marked with a bug icon.
  - **Input that defines the test case:** For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument.
    If input from a stub caused the error, stub information will be displayed here. Empty boxes indicate automatically generated stubs. Black boxes indicate user-defined stubs. To see the stack trace where a stub invocation occurred, expand the stub's branch. For more information on stubs, see "Testing Classes That Reference External Resources" on page 93 and "Using Custom Stubs" on page 98.
    Marked with an arrow icon.


error
input
[2] Specification and Regression Errors: done
add: RETVAL = 3 (before was 0)
Test Case Input
int RETVAL = examples.eval.Simple.add (0, 0);

Exploring and Customizing Class Test Results

The Class Testing UI provides you with a variety of ways to explore test results and to customize what results are reported the next time the test is run. Some actions that you might want to perform when viewing results include:

- **View/evaluate test cases:** To view and/or evaluate the automatically-generated and user-defined test cases used to test this class, click the **View** tool bar button.
- **View the source responsible for a rule violation or exception:** To view the source of the error, with the problematic line highlighted, double-click the file/line information for the error in the Errors Found panel.
- **Edit your class:** To open your class in a text editor, right-click the **Source** button, then choose **Edit Source** from the shortcut menu.
- **View a description of a violated rule:** To view a description of a violated static analysis rule, along with an example of that rule and a suggested repair, right-click a static analysis error message that has a wizard hat or bug icon, then choose **View Rule Description** from the shortcut menu.
- **View the stack trace of an uncaught runtime exception:** To view a stack trace like the one that the Java virtual machine would give if a reported uncaught runtime exception were thrown open the branch containing the uncaught runtime exception.
- **View the calling sequence of an uncaught runtime exception:** To view an example usage of the class that leads to the reported uncaught runtime exception, open any uncaught runtime exception's **Test Case Input** branch.
- **View the error-causing input:** To view the error-causing input, open any specification or regression testing error or uncaught runtime exception error's **Test Case Input** branch.

- **View an example test case:** To view an example Java program that executes the input for a test case, right-click the **Test Case Input** node, then choose **View Example Test Case** from the shortcut menu.
  If an exception has been reported for this input, the exception will be thrown when you run this program.
  Sometimes (for example, while testing an abstract class) the input that Jtest finds doesn't correspond to a compilable Java program.
  If the input includes stubs, the generated .java program will include only the stub text.
- **View a report:** To view a report file, click **Report**.
- **View metrics:** To view class metrics, click **Metrics**.
- **Gauge coverage:** There are two ways to gauge test coverage:
  - Review the coverage data displayed in the Test Progress panel.
  - Display and review the report file.

    The report file contains, among other information, the annotated source code for the tested class. This may be used to determine what lines Jtest tested and what lines it did not test.
    A method is designated "covered" if Jtest automatically tests any part of the constructor. Jtest also reports branch coverage. For example, if a piece of code has an if statement, there are two branches; if one branch is covered (the true path), Jtest reports 50% coverage of that branch.
- **Modify test case evaluation:** If a reported uncaught runtime exception is actually the correct behavior of the class, or if you want Jtest to ignore the outcome of an input while checking for specification and regression errors, right-click the error message with the bug icon, then choose the appropriate command from the shortcut menu.
  - To indicate that a reported error is not an error, choose **Not an Error**.

    - To tell Jtest to ignore the outcome for this input, choose **Ignore this Outcome**. If you choose this option, the outcome will not be used for comparisons when searching for specification or regression errors. Also, no uncaught runtime exceptions will be reported for this test case input.
    - To have Jtest ignore all outcomes in a class's **Uncaught Runtime Exceptions** or **Specification and Regression Errors** node, or to indicate that all errors contained in a class's **Uncaught Runtime Exceptions** or **Specification and Regression Errors** node are not actual errors, right-click the appropriate node and choose **Set All to: Not an Error** or **Set All to: Ignore this Outcome**.
- **Suppress messages:** To suppress the reporting of a single, specific exception or static analysis violation, right-click the message (with the bug icon) related to the error/violation that you do not want reported in future test runs, then choose **Suppress** from the shortcut menu. This automatically adds the suppression to the appropriate Suppressions Table (for dynamic analysis messages) or Suppressions List (for static analysis messages).

Testing a Set of Classes

In Jtest's Project Testing UI, you can automatically test all (or some) of the classes contained in any directory, jar file, or zip file with a single click. Jtest automatically searches the specified directory, jar file, or zip file, and tests all of the classes that it finds.

It is possible to perform all testing-related activities in the Project UI. The Project Testing UI contains all results for all classes tested, and allows you to access the same features that are available in the Class Testing UI. If you want to test an entire project, then focus on results on a class-by-class basis, you can test the project in the Project Testing UI, then open the class(es) that you want to focus on in the Class Testing UI.

By default, the Project Testing UI performs dynamic analysis only on public classes; this setting can be changed in the **Search Parameters> Dynamic Analysis** branch of the Project Test Parameters tree.

Also by default, Jtest will not test a class that it has previously tested unless that class has been modified since the previous test. Jtest determines whether or not a class has changed by checking that both the .class file and the .java file contents have not changed. Timestamps are not considered. To force Jtest to test all classes on every test, disable the **Skip Classes Already Tested** node in the **Search Parameters** branch of the Project Test Parameters tree.

To test a set of classes:

1. Open the Project Testing UI by clicking the **Project** button in the Class Testing UI tool bar, or by choosing **Window> Project Testing UI** in the Class Testing UI menu bar.
2. In the **Search In** field of the Project Testing UI, specify what directory, zip file, .jar file, or set of files you want Jtest to test.
    - To browse for a directory, jar file, or zip file that you want Jtest to start searching and testing, click the **Browse** button. To select several files at once, CTRL-click or SHIFT-click to select the files that you want to test.

    If the parameter is a directory, Jtest will recursively traverse the path's subdirectories, zip files, and jar files, searching for and testing any classes it finds.

If the parameter is a jar or zip file, Jtest will open the file and search it for classes in which to find errors.

3. (Optional) If you want to restrict the classes that Jtest tests, do one of the following:

    - Use the **Filter-in** field to tell Jtest to find and test only classes that match the given expression. Use regular expressions to indicate what types of files to include.

      For example, if you want Jtest to look only for classes in the util package, enter the following parameter in this field

      ```
      util.*
      ```

      When this field is left empty, all classes found will be tested.
      For more information about entering regular expressions in this field, see “Controls Panel” on page 218.

    - Use the Skip List to indicate specific classes that you want Jtest to skip. The Skip List is accessible by clicking the **Skip List** node in the **Search Parameters** branch of the Project Test Parameters panel. Clicking this node opens a dialog box that lets you enter the names of specific classes in your project that you do not want tested.

    - Use the Test Only List to indicate specific classes that you want Jtest to test. The Test Only List is accessible by clicking the **Test Only List** node in the **Search Parameters** branch of the Project Test Parameters panel. Clicking this node opens a dialog box that lets you enter the names of specific classes in your project that you want tested.

4. Start the test by clicking the **Start** tool bar button.

    - If you only want to perform static analysis or a specific type of static analysis, right-click the **Start** button and choose the menu item that describes the type of test that you want to perform.

    - If you only want to perform dynamic analysis or a specific type of dynamic analysis, right-click the **Start** button and

choose the menu item that describes the type of test that you want to perform.

Unless you tell it to do otherwise, Jtest automatically performs all steps required for:

- Static analysis
- White-box testing
- Black-box testing (if Design by Contract-format contract information is included in the classes under test or test case outcomes have been validated)
- Regression testing (on all test runs after the first)

If you want to configure Jtest to perform only static analysis or only dynamic analysis, modify your testing parameters as described in “Customizing Test Parameters” on page 180.

To have Jtest stop finding and testing classes, click the **Stop** button.

To have Jtest temporarily pause (or resume) finding and testing classes, click the **Pause** button. If you pause testing, Jtest will finish testing the current class before pausing.

- “About Static Analysis” on page 69
- “About Dynamic Analysis” on page 85
- “About White-Box Testing” on page 108
- “About Black-Box Testing” on page 113
- “About Regression Testing” on page 152

Results

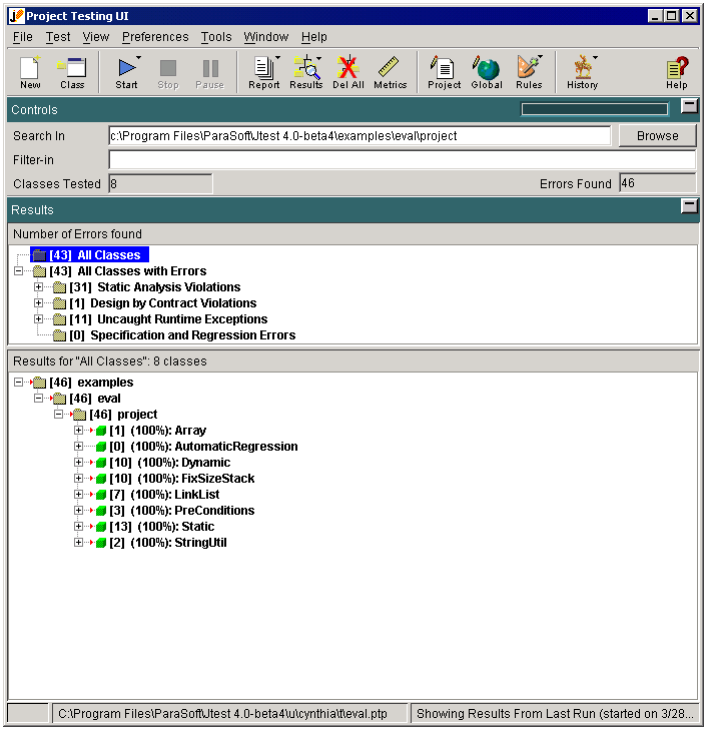
Results are displayed in the Project Testing UI's Results panel. To learn more about this panel's branches and available options, see “Understanding the Results Panel” on page 45 and “Exploring and Customizing Project Test Results” on page 53.

Testing a Set of Classes - Example

The following example demonstrates the completely automatic mode of testing a set of classes.

1. Open Jtest.
2. Click the **Project** button in the Class Testing UI tool bar. The Project Testing UI will open.
3. Click the **Browse** button in the Project Controls panel, locate the Jtest white-box testing examples directory (<jtest_install_dir>/examples/eval/project) in the file chooser that opens, then click **Open**.
4. Click the **Start** button in the Project UI tool bar.

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



Results are displayed in the Project Testing UI's Results panel. To learn more about this panel's branches and available options, see “Understanding the Results Panel” on page 45 and “Exploring and Customizing Project Test Results” on page 53.

Related Topics

“Jtest Tutorials” on page 268

Understanding the Results Panel

All results from a project test are displayed in the Project Testing UI's Results panel after the test is completed. You can load results from previous tests into this panel by clicking the **Results** button.

The contents of this panel are described below, and in context-sensitive help. If commands are available by right-clicking a particular node, a right-click icon will open when you place your cursor over that node.

The Project Testing UI also provides you with a variety of ways to gain additional information about the tests performed and to customize what results are reported the next time that this test is run. For more information about these options, see "Exploring and Customizing Project Test Results" on page 53.

The Results panel contains two windows: the Number of Errors Found Window and the Results for All Classes Window.

Number of Errors Found Window

This window displays the distribution of the errors found. The tree in this window contains a node for every type of error that Jtest detects, along with the number of that type of error found in the project test.

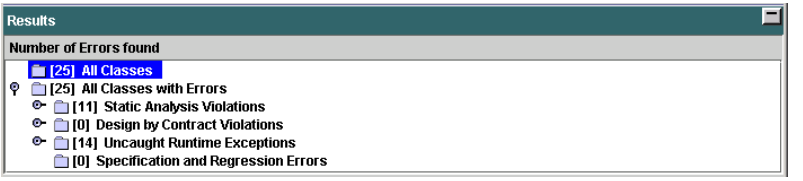
This window lets you determine what results are displayed in the Results For All Classes window. To make the lower results window display only a certain type of result (such as All Classes With Errors, Uncaught Runtime Exceptions, or java.lang.NullPointer Exceptions) perform the following steps:

1. In the Number of Errors Found window, right-click the node that describes the type of results you want to view. A shortcut menu will open.

   Note: If you do not see a node representing the type of result that you want to view, expand the **All Classes With Errors** tree branches.

2. Choose **Show Results for this category** from the shortcut menu.

The node that describes the types of results displayed in the lower Results window will be highlighted in blue.

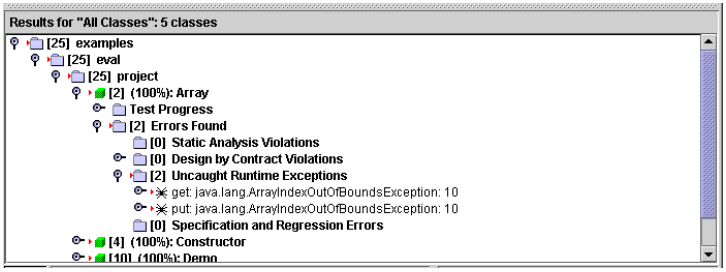


Results for All Classes Window

This window lists the results for the current project test. The results are organized into a tree. Each tree branch corresponds to the results for one class. The block next to each class name indicates class properties as follows:

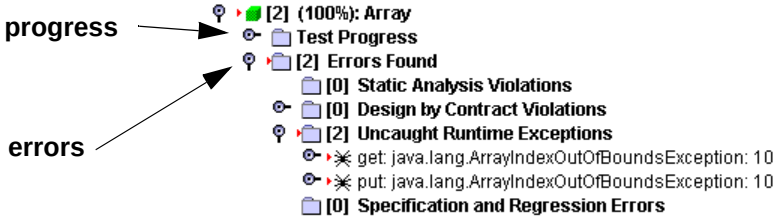
- Red: private class
- Orange: protected class
- Green: public class
- Blue: package-private class

To view results for a class, expand the branches that correspond to that class.



Each class has two main sub-branches:

- **Test Progress:** Contains information about test status and coverage.
- **Errors Found:** Contains information about errors found.



Test Progress

The Test Progress branch contains the following information:

- **Static Analysis:** Displays the progress of static analysis tests. While static analysis is being performed, a percentage indicating test progress is displayed to the right of this node. When a test is complete, the word “done” will appear to the right of this node

  The **Number of Rules Analyzed** node displays the number of static analysis rules analyzed.

- **Dynamic Analysis:** Displays the progress of dynamic analysis tests. While dynamic analysis is being performed, a percentage indicating test progress is displayed to the right of this node. When a test is complete, the word “done” will appear to the right of this node.

  Dynamic coverage is shown only for classes on which Jtest has performed dynamic analysis. By default, dynamic analysis is only performed on the public classes; static analysis is performed on all classes found (public and non-public).

  The **Number of Test Cases Executed** node displays the total number of test cases executed. These test cases are divided into two categories: automatic and user-defined. The **Automatic** node displays the number of automatically-generated test cases executed. The **User Defined** node displays the number of user-defined test cases executed.

  The **Number of Outcome Comparisons** node displays the number of outcomes compared during black-box and regression testing.

  The **Total Coverage** node displays the cumulative coverage that Jtest achieved.

  Jtest performs data coverage for the generated input categories; this means that the parts of the class that have been covered are thoroughly tested with respect to those inputs. The coverage reported is relative to the classes that have been accessed for the paths Jtest has tried. If some part of the class is not covered, it means that Jtest has not yet found a path leading to those statements or no path leads to those statements. In class testing mode, Jtest usually covers approximately 50% of a class's code. Sometimes Jtest will be able to test 100% of the class, and sometimes it will test less than 50% of the class.

  The **Total Coverage** branch’s **Multi-condition branch** node dis-

plays coverage achieved on branches. A branch is a path of execution through the statements. Selection statements, such as “switch” and “if” have one or more branches per statement. Branch coverage is a measure of what percentage of branches were covered given the total number of branches in the code.

The **Total Coverage** branch’s **Method** node displays coverage achieved on methods. Method coverage is a measure of what percentage of methods were covered given the total number of methods in the code.

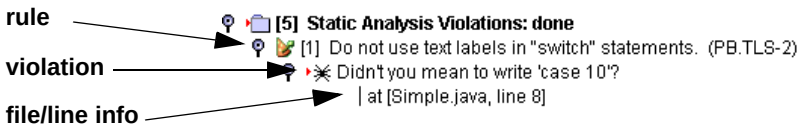
The **Total Coverage** branch’s **Constructor** node displays coverage achieved on constructors. Constructor coverage is a measure of what percentage of constructors were covered given the total number of constructors in the code.

Errors Found

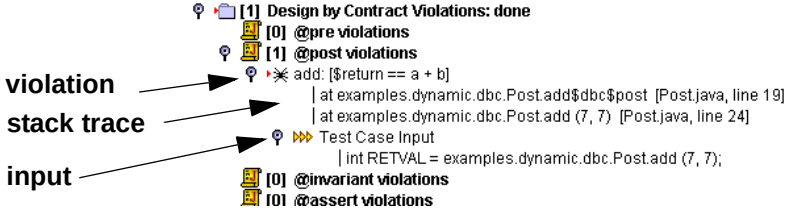
The Errors Found branch is organized like the Errors Found tree in the Class Testing UI Errors Found panel. It contains the following information:

Note: All error/violation messages are marked with a bug icon.

- **Static Analysis Violations:** Displays the number of violations that Jtest found while performing static analysis. This branch contains the following information:
    - **Rule:** Name of rule violated. (Rule ID is displayed in parentheses).
      Marked with a wizard hat icon.
    - **Violation:** Jtest rule violation message. To suppress this message or view the associated rule description, right-click this node then choose the appropriate command from the shortcut menu.
      Marked with a bug icon.
    - **File/line info:** File/line number where violation occurred. To view or edit the source code, right-click this node then choose the appropriate command from the shortcut menu.



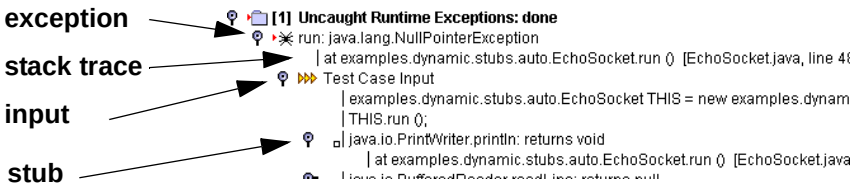
- **Design by Contract Violations:** Displays the number of Design by Contract violations that Jtest found while performing dynamic analysis. Design by Contract violations are organized according to the nature of the violation. This branch contains the following violation categories:
  - **@pre violations:** Contains information about violations that occur when a method is called incorrectly.
  - **@post violations:** Contains information about violations that occur when a method does not return the expected value.
  - **@invariant violations:** Contains information about violations that occur when an @invariant contact condition is not met.
  - **@assert violations:** Contains information about violations that occur when an @assert contact condition is not met.



- **Uncaught Runtime Exceptions:** Displays the number of uncaught runtime exceptions that Jtest found while performing dynamic analysis. Each uncaught runtime exception is followed

by a full stack trace, as well as an example input leading to this exception. This branch contains the following information:

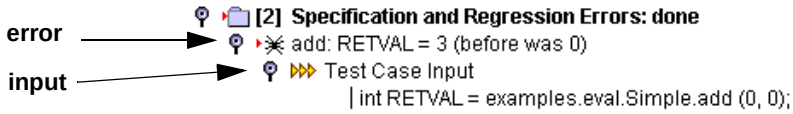
- **Exception:** Exception found.
  Marked with a bug icon.
- **Stack trace information:** A stack trace like the one that the Java virtual machine would give if a reported uncaught runtime exception were thrown. To view or edit the source code, right-click this node then choose the appropriate command from the shortcut menu. (If the file and line number information is missing, recompile the class with debug information).
- **Input that defines the test case:** For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument.
  If input from a stub caused the exception, stub information will be displayed here. Empty boxes indicate automatically generated stubs. Black boxes indicate user-defined stubs. To see the stack trace where a stub invocation occurred, expand the stub's branch. For more information on stubs, see "Testing Classes That Reference External Resources" on page 93 and "Using Custom Stubs" on page 98.
  Marked with an arrow icon.



- **Specification and Regression Errors:** Displays the specification and regression errors that Jtest found while performing dynamic analysis. The errors are determined by comparing the test case outcomes of this run with those of previous runs or

those specified by the user. (To view the reference outcomes, open the Class Test Parameters window and browse to **Dynamic Analysis> Test Case Evaluation> Specification and Regression Test Cases**). This branch contains the following information:

- **Error:** Specification and regression error found.
  Marked with a bug icon.
- **Input that defines the test case:** For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument.
  If input from a stub caused the error, stub information will be displayed here. Empty boxes indicate automatically generated stubs. Black boxes indicate user-defined stubs. To see the stack trace where a stub invocation occurred, expand the stub's branch. For more information on stubs, see "Testing Classes That Reference External Resources" on page 93 and "Using Custom Stubs" on page 98.
  Marked with an arrow icon.



Exploring and Customizing Project Test Results

The Project Testing UI provides you with a variety of ways to explore test results, as well as to customize what results are reported the next time this test is run. Most options are accessed via shortcut menus in the lower Results window. Some actions that you might want to perform when viewing results include:

- **View/evaluate test cases:** To view and/or evaluate the automatically-generated and user-defined test cases used to test this class, right-click either the **[Class Name]** node, or the **[Class Name]> Test Progress> Dynamic Analysis> Number of Test Cases Executed** node, then choose **View Test Cases** from the shortcut menu.
- **View the source responsible for a rule violation or exception:** To view the source of the error, with the problematic line highlighted, double-click the file/line information for any error contained within the **Errors Found** branch.
- **View a description of a violated rule:** To view a description of a violated static analysis rule, along with an example of that rule and a suggested repair, right-click a static analysis error message with a wizard or bug icon, then choose **View Rule Description** from the shortcut menu.
- **View the stack trace of an uncaught runtime exception:** To view a stack trace like the one that the Java virtual machine would give if a reported uncaught runtime exception were thrown, open the uncaught runtime exception's branch.
- **View the calling sequence of an uncaught runtime exception:** To view an example usage of the class that leads to the reported uncaught runtime exception, open the uncaught runtime exception's **Test Case Input** branch.
- **View error-causing input:** To view the error-causing input, open any specification or regression testing error or uncaught runtime exception error's **Test Case Input** branch.

- **View an example test case:** To view an example Java program that executes the input for a test case, right-click the **Test Case Input** node, then choose **View Example Test Case** from the shortcut menu.

  If an exception has been reported for this input, the exception will be thrown when you run this program.

  Sometimes (for example, while testing an abstract class) the input that Jtest finds doesn't correspond to a compilable Java program.

  If the input includes Stubs, the generated .java program will include only the stub text.
- **View a report:** To view a report file, click the **Report** button in the Project Testing UI tool bar.
- **View metrics:** To view project and average class metrics, click **Metrics**. To view a specific class's metrics, right-click that class's node in the Results panel, then choose **View Class Metrics**.
- **Gauge coverage:** To review coverage, open the **[Class Name]> Test Progress> Dynamic Analysis> Total Coverage** node. A method is designated "covered" if Jtest automatically tests any part of the constructor. Jtest also reports branch coverage. For example, if a piece of code has an "if" statement, there are two branches; if one branch is covered (the true path), Jtest reports 50% coverage of that branch.
- **Modify test case evaluation:** If a reported dynamic analysis error is actually the correct behavior of the class, or if you want Jtest to ignore the outcome of an input while checking for specification and regression errors, right-click the error message with the bug icon, then choose the desired command from the shortcut menu.
  - To indicate that a reported error is not an error, choose **Not an Error**.
  - To tell Jtest to ignore the outcome for this input, choose **Ignore this Outcome**. If you choose this option, the out-

come will not be used for comparisons when searching for specification or regression errors. Also, no uncaught runtime exceptions will be reported for this test case input.

To have Jtest ignore all outcomes in a class's **Uncaught Runtime Exceptions** or **Specification and Regression Errors** node, or to indicate that all errors contained in a class's **Uncaught Runtime Exceptions** or **Specification and Regression Errors** node are not actual errors, right-click the appropriate node and choose **Set All to: Not an Error** or **Set All to: Ignore this Outcome**.

- **Suppress messages:** To suppress the reporting of a single, specific exception or static analysis violation, right-click the error message that you do not want reported in future test runs, then choose **Suppress** from the shortcut menu. This automatically adds the suppression to the appropriate Suppressions Table (for dynamic analysis messages) or Suppressions List (for static analysis messages).
- **Remove a class's results from the Results panel and Results folder:** To remove the results of a class from both the Results panel and the Results Folder (which stores project test results), right-click the **[Class Name]** node, then choose **Delete** from the shortcut menu.
- **Edit Class Test Parameters:** To modify a specific class's Class Test Parameters, right-click the **[Class Name]** node, then choose **Edit Class Test Parameters** from the shortcut menu.
- **Load in Class Testing UI:** To focus on the errors for a single class, view the class in the Class Testing UI by right-clicking the **[Class Name]** node, then choosing **Load in Class Testing UI** from the shortcut menu.

Loading One of a Project's Classes in the Class Testing UI

There are three ways to open one of a project's classes in the Class Testing UI:

Method 1

Before or After a Project Test

1. In the Project Testing UI, click **Project** to open the Project Test Parameters. The Project Test Parameters window will open.
2. In the Project Test Parameters window, open the **Classes in Project** branch.
3. Right-click the node that corresponds to the class that you want to open in the Class Testing UI. A shortcut menu will open.
4. Choose **Load Class in Class Testing UI** from the shortcut menu. The class and its test parameters will then be loaded in the Class Testing UI.

Method 2

After a Project Test Including the Class You Want to Open

1. If the Project Testing UI's Results panel does not contain test results, open the results.
2. In the lower Results panel, right-click the node whose name corresponds to the class that you want to open in the Class Testing UI. A shortcut menu will open.

3. Choose **Load in Class Testing UI** from the shortcut menu. The class and its test parameters will be loaded in the Class Testing UI.

Method 3 (In the Class Testing UI)

- In the Class Testing UI, choose **File> Open** and browse to the “ctp” class test parameters file associated with the class that you want to open. The class and its test parameters will be loaded in the Class Testing UI.

Editing Class Test Parameters from the Project Testing UI

If you are testing a project and want to add user-defined test cases or change class-specific test parameters, you must edit the appropriate class's Class Test Parameters.

There are two ways to edit Class Test Parameters from the Project Testing UI:

Method 1

Before or After a Project Test

1. In the Project Testing UI, click **Project** to open the Project Test Parameters. The Project Test Parameters window will open.
2. In the Project Test Parameters window, open the **Classes in Project** branch.
3. Right-click the node that corresponds to the class whose parameters you want to modify. A shortcut menu will open.
4. Choose **Edit Class Test Parameters** from the shortcut menu. The Class Test Parameters window will open.
5. Modify parameters in the Class Test Parameters window.

Method 2

After a Project Test Including the Class Whose Parameters You Want to Modify

1. If the Project Testing UI's Results panel does not contain test results, open the results.

2. In the lower Results panel, right-click the node whose name corresponds to the class whose parameters you want to modify. A shortcut menu will open.
3. Choose **Edit Class Test Parameters** from the shortcut menu. The Class Test Parameters window will open.
4. Modify parameters in the Class Test Parameters window.

Saving Your Changes

If you start a new test or exit the Project Testing UI after changing class test parameters through the Project Testing UI, Jtest will automatically ask you if you want to save the changes that you made. If you would like to save your modified class test parameters before that, perform the following steps:

1. In the lower Results panel, right-click the node whose name corresponds to the class whose parameters you want to save. A shortcut menu will open.
2. Choose **Save Current Changes** from the shortcut menu.

Jtest will them save that class's parameters in <jtest_install_dir>/u/<username>/jtest.properties.

Running Jtest in Batch Mode

You can run an existing test from the command line using existing .ctp or .ptp test parameter files, or you can test a class or project for the first time and have Jtest create all necessary parameter files.

Windows

To run Jtest in batch mode on a Windows system:

1. Set up command line Jtest.
    a. Change directories to the Jtest installation directory.
    b. Run the `jtvars.bat` program by entering the following command at the prompt:

        ```
        jtvars.bat
        ```
2. Run the test from the command line by entering your command at the prompt. For tips on creating commands, see “Available Command Line Options” and “Example Commands” below.

After the test is complete, you can view the report that was generated, or-- if you tested a project-- you can view the results within the Jtest UI. To view results within the UI, launch Jtest, then open the .ptp file for the project that you tested in batch mode. Click the **Results** button to display the results. After the results are loaded in the Results panel, you can also generate additional types of reports by right-clicking the **Report** button and selecting the type of report that you want to see.

UNIX

To run Jtest in batch mode on a UNIX system:

- Run the test from the command line by entering your command at the prompt. For tips on creating commands, see “Available Command Line Options” and “Example Commands” below.

After the test is complete, you can view the report that was generated, or-- if you tested a project-- you can view the results within the Jtest UI. To view results within the UI, launch Jtest, then open the .ptp file for the project that you tested in batch mode. Click the **Results** button to display the results. After the results are loaded in the Results panel, you can also generate additional types of reports by right-clicking the **Report** button and selecting the type of report that you want to see.

Available Command Line Options

Available command line options for `jtestgui` include:

Command	Meaning
`-help`	Print out list of available options.
`-nogui`	Run jtestgui without the UI. `jtestgui -nogui Simple.ctp`
`-nolog`	Do not generate log messages to `stdout`.
`-nologo`	Do not show logo message.
`-filter_in <regexp>`	Set Project Test Parameter’s Filter-in field value. To test only the examples/eval directory use: `jtestgui -filter_in examples\eval\ -test examples.ptp`

`-run_only <what>`	Run specific types of tests only. Options for `<what>` include: • `static`: Runs static tests. • `static_builtin`: Runs all built-in static analysis rules. • `static_user`: Runs all user-defined rules. • `static_<XXX>`: Runs only rules in the `XXX` category; for example, use `static_UC` to run only “Unused Code” rules. • `dynamic`: Runs dynamic tests • `dynamic_auto`: Runs automatic test cases • `dynamic_user`: Runs user-defined test cases. Use `-run_only help` to see a list of all options for `<what>`.
`-ctp <file>.ctp`	Load class test parameters file <file>.ctp.
`-ptp <file>.ptp`	Load project test parameters file <file>.ptp.

`-ctp_new <file>.ctp`	Create and load class test parameters in the file <file>.ctp. If file <file>.ctp already exists, it will be overwritten by this new file. Use the following options when creating a new .ctp file from the command line: • `-class_name <name>`: Sets the class test parameters' "Class Name" parameter. Indicates which class you want to test. • `-cp`: Sets the class test parameters' -cp parameter. Should specify the fully-qualified name of the path. • `-classpath`: Sets the class test parameters' -classpath parameter. • `-sourcepath`: Sets the class test parameters' -sourcepath parameter.

`-ptp_new <file>.ptp`	Create and load project test parameters in the file <file>.ptp. If file <file>.ptp already exists, it will be overwritten by this new file. Use the following options when creating a new .ptp file from the command line: • `-search_in <dir|jar|zip>`: Sets the project test parameters' "Search In" parameter. Indicates which set of classes you want to test. • `-filter_in <dir|jar|zip>`: Sets the project test parameters' "Filter In" parameter. Indicates which particular classes you want to test. • `-cp`: Sets the project test parameters' -cp parameter. Should specify the fully-qualified name of the path. • `-classpath`: Sets the project test parameters' -classpath parameter. • `-sourcepath`: Sets the project test parameters' -sourcepath parameter.
`-gtp <filename>`	Use Global Test Parameters in `<filename>`. If <filename> doesn't exist, Jtest creates one with default values. `jtestgui -gtp newProjects.gtp`

`-report[_ascii|_html] <filename|std-out|stderr>`	Generate report in `<filename>`. Needs to be used with `-test`. `jtestgui -report report.txt` **Note:** If you don't specify `ascii` or `html`, Jtest uses the format specified in your Global Test Parameters.
`-summary_report[ascii|html] <filename|std-out|stderr>`	Generate summary report in `<file-name>`. Needs to be used with `-test` `jtestgui -summary_report report.txt -test Simple.ctp`
`-detail_report[ascii|html] <filename|std-out|stderr>`	Generate detail report in `<filename>` needs to be used with `-test`. `jtestgui -detail_report report.txt -test Simple.ctp`
`-retest`	Force retesting of all classes-- even classes tested in a previous run **Note:** This option is similar to the **Skip classes already tested** option in the Project Test Parameters. It applies to .ptp files and is used for all tests run during that jtestgui invocation. `jtestgui -retest`
`-silent`	Only generate output if errors are found. Project reports only contain
`-ruledir`	Overwrites the Global Test Parameters' rules directory parameter (specifies the location of your user-defined rules). Used only in silent mode. `jtestgui -nogui -ruledir <directory name> -ptp test.ptp`

To see all available command-line options, enter

```
jtestgui -help
```

at the prompt.

Note: If you are on a Windows system, you must set the environment for command line Jtest before you enter this command.

Example Commands

- To run the previously tested MyTest test whose test parameters are saved in c:\progra~1\parasoft\jtest\tests\MyTest.ptp, then generate a summary report (summary.rpt) in the current working directory, enter the following command at the prompt:

  ```
  jtestgui -summary_report summary.rpt -ptp c:\pro-
  gra~1\parasoft\jtest\tests\MyTest.ptp
  ```

- To perform the above test in “silent mode” (without displaying the Jtest UI), you enter the following command at the prompt:

  ```
  jtestgui -nogui -summary_report summary.rpt -ptp
  c:\progra~1\parasoft\jtest\tests\MyTest.ptp
  ```

- To test a single class, use the -ctp flag instead of the -ptp flag, then enter the path to the .ctp file instead of the path to the .ptp file. For example, to run the previously tested MyTest class whose test parameters are saved in /usr/users/carson/jtest/tests/MyClass.ctp, then generate a summary report (summary2.rpt) in the current working directory, enter the following command at the prompt:

  ```
  jtestgui -summary_report summary2.rpt -ctp
  /usr/users/carson/jtest/tests/MyClass.ctp
  ```

- To test a class that has never been tested and that has class dependencies, use the -ctp_new, -class_name, -cp, -classpath, and -sourcepath tags. For example, to create a .ctp file for a class named Foo.class in C:\Jtest, enter the following command at the prompt:

```
jtestgui -ctp_new <name of ctp file> -class_name <path
to Foo.class> -cp C:\Jtest -sourcepath < path to
source code if not in same dir as Foo.class>
```

When this command is executed, Jtest will open with these parameters set.

- To test a project that has never been tested before, you can use the -ptp_new flag in conjunction with the -search_in flag to specify the directory, jar, or zip that you want to test. In addition, if only certain classes should be tested in the project, the -filter_in flag can be used to indicate what classes you want to test and the -cp, -classpath, and -sourcepath flags can be used to set paths to dependencies. For example, to set up a .ptp file for Foo.jar where only some classes are tested, enter the following command at the prompt:

  ```
  jtestgui -ptp_new < name of ptp file > -search_in <
  path the Foo.jar > -filter_in < regular expression
  specifying the files to be tested > -cp < full path to
  dependencies > -sourcepath < path to source code>
  ```

  Note: Jtest is unable to find source code that is located within a jar file. For this example, the -sourcepath would point to the source code of the project that has been unjarred.

Testing a Large Project

If you are testing a large project (over 1000 classes), we recommend that you test in batch mode, and that you perform several smaller tests rather than one large one.

Each test should test a package or a tree of packages.

To break the test run into smaller tests, use the Project Testing UI's **Filter-in** field to define your smaller tests. For example, you could define three tests using com.tech.util.*, com.tech.tool1.*, and com.tech.tool2.*, then save these tests as util.ptp, tool1.ptp, and tool2.ptp.

To run these tests, you would invoke `jtestgui` one time for each small test that you want to run. For example, assuming that you want to run the tests referenced in the previous paragraph, you could use the following 3 commands:

```
jtestgui -nogui -ptp util.ptp -summary_report util.rep
jtestgui -nogui -ptp tool1.ptp -summary_report tool1.rep
jtestgui -nogui -ptp tool2.ptp -summary_report tool2.rep
```

About Static Analysis

During static analysis, Jtest analyzes source code to expose violations of coding standards, language specific “rules” that help you prevent errors.

Jtest enforces the following types of coding standards:

- **Traditional coding standards:** Traditional coding standards are rules that apply to constructs within the class under test. A traditional coding standard might test whether or not a file’s source code contains a construct that has a high probability of resulting in an error. For example, one traditional coding standard checks that you use “equals” instead of “==” when comparing strings (writing “==” when you should have used “equals” causes the program to check if two strings are identical, rather than check if two strings have the same value).
- **Global coding standards:** Global coding standards are rules that ensure that projects use fields, methods, and classes wisely. A global coding standard might check that a project does not contain logical flaws and unclear code such as unused or overly-accessible fields, methods, and classes; such problems increase the probability of an error being introduced into the code, and may also make the code less object-oriented.
- **Metrics:** Metrics are measures of the size and complexity of code. When Jtest performs static analysis, it also measures your class’s and (if applicable) project’s metrics; it reports all metrics in the Metrics window, and reports a static analysis violation for any class metric that is outside of the bounds that have been set for that metric.
- **Custom coding standards:** Custom coding standards are rules specially tailored to your own or your group’s unique coding style. For information on creating custom coding standards, refer to “Creating Your Own Static Analysis Rules” on page 83.

Static analysis is performed automatically when you test a class or project.

When enforcing all static analysis rules *except for Global Static Analysis rules*, Jtest statically analyzes the classes by parsing the .java source and applying a set of rules to it, then alerting you to any rule violations found.

When enforcing global static analysis rules, Jtest scans all of the project's .class files to collect global usage information, uses this information to check if each class violates any rules in Jtest's Global Static Analysis category, then alerts you to any violations in the classes under test. Because Jtest uses .class files (rather than .java files) to check this particular category of rules, you can perform global static analysis even when .java files are not available.

Note: Global static analysis can only be performed while testing a project.

Related Topics

"Performing Static Analysis" on page 71

"Creating Your Own Static Analysis Rules" on page 83

"Viewing Class and Project Metrics" on page 73

"Tracking Metrics Over Time" on page 76

"Customizing Static Analysis" on page 79

Performing Static Analysis

Jtest performs static analysis, along with all other appropriate types of testing, each time that you test a class or set of classes. Traditional static analysis (checking static analysis rules that are *not* in the Global Static Analysis category) can only be performed on classes whose .java source files are available. Global static analysis can be performed as long as .class files are available.

To perform static analysis:

1. Open the appropriate UI for your test. The Class Testing UI is used to test a single class; the Project Testing UI is used to test a set of classes.
    - 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. Use the **Browse** button to indicate what class or set of classes you want to test.
4. Test the class or project by clicking the **Start** button.
    - If you only want to perform static analysis, right-click the **Start** button, then choose **Static Analysis** from the short-cut menu.
    - If you only want to enforce a particular set of rules, right-click the **Start** button, then choose **Dynamic Analysis> <type of rules you want enforced>** from the short-cut menu.

Jtest will then run all requested tests.

Uncaught runtime exceptions found will be reported in the **Static Analysis Violations** branch of the Errors Found Panel (if you tested a single class) or the Results Panel (if you tested a project).

A description of each rule, as well as an example violation and suggested repair, appears in the Rules section of the Reference Guide. You can also see a description of a rule by right-clicking the rule's violation message (the line with the bug icon), then choosing **View Rule Description** from the shortcut menu.

Related Topics

"About Static Analysis" on page 69

"Creating Your Own Static Analysis Rules" on page 83

"Viewing Class and Project Metrics" on page 73

"Tracking Metrics Over Time" on page 76

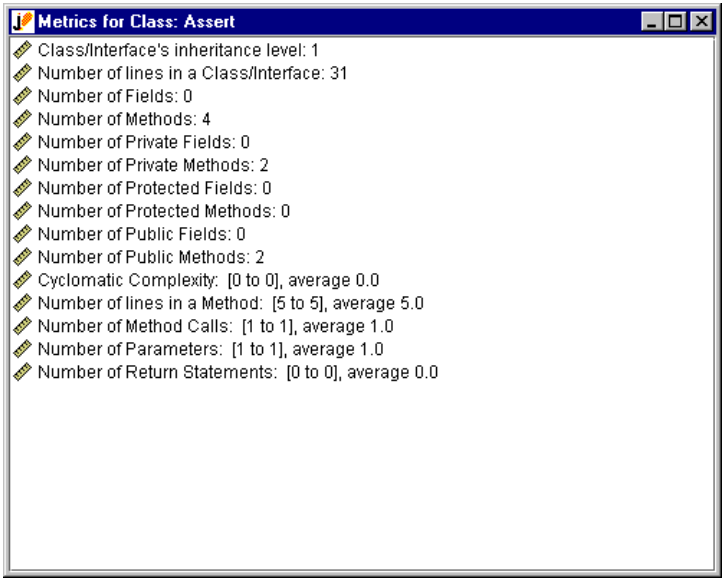
"Customizing Static Analysis" on page 79

"Jtest Tutorials" on page 268

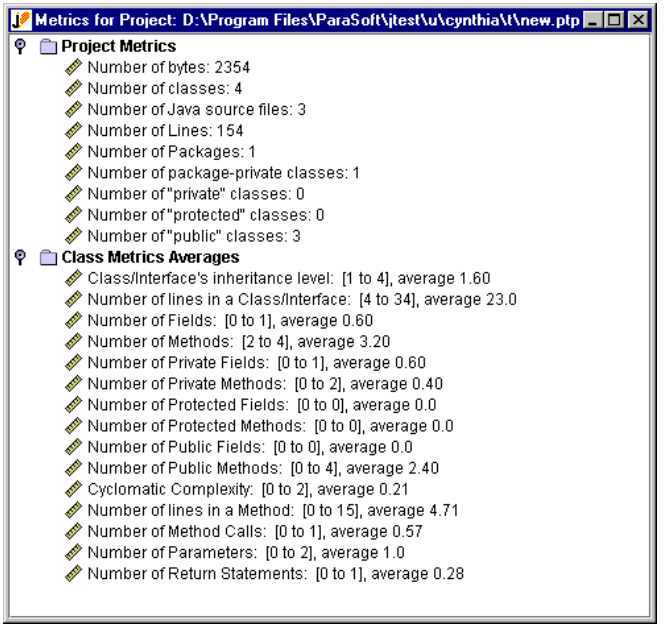
Viewing Class and Project Metrics

Jtest automatically measures metrics when it performs static analysis. If any of your metrics are out of the suggested or customized “legal” bounds, Jtest will report a static analysis violation for each out-of-bound metric.

Jtest also offers a summary of all class and project metrics. You can view this summary by clicking the **Metrics** tool bar button. The Class Testing UI’s metrics window will contain metrics for the class; the Project Testing UI’s metrics window will contain both project metrics (metrics about aspects of the project) and class metrics averages (the average class metrics of all of the tested classes).



Metrics for a Class



Metrics for a Project

To view class metrics for a class tested as part of a project, right-click that class's node in the Project Testing UI's Results panel, then choose **View Class Metrics** from the shortcut menu.

To print your metrics, control-right-click the unused area of the Metrics window, then choose **Print** from the shortcut menu.

To view a full description of a project or class metric, right-click the metric that you want to learn more about, then choose **View Rule Description** from the shortcut menu that opens.

Jtest can also graph how your project metrics change over time. For information on graphing metrics, see "Tracking Metrics Over Time" on page 76.

Related Topics

“About Static Analysis” on page 69

“Performing Static Analysis” on page 71

“Creating Your Own Static Analysis Rules” on page 83

“Tracking Metrics Over Time” on page 76

“Customizing Static Analysis” on page 79

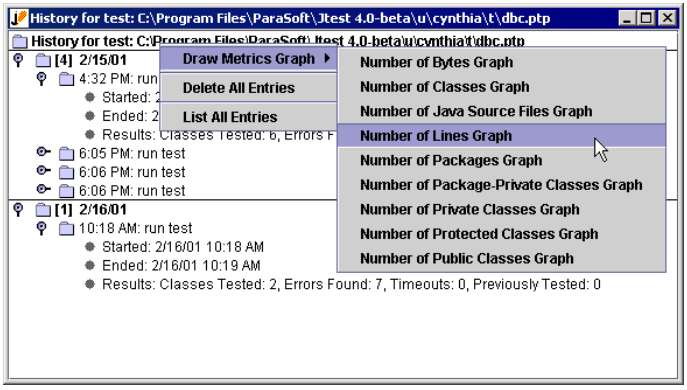
Tracking Metrics Over Time

Jtest saves your project metrics for each test and can graph how each metric changes over time. You can graph the following metrics:

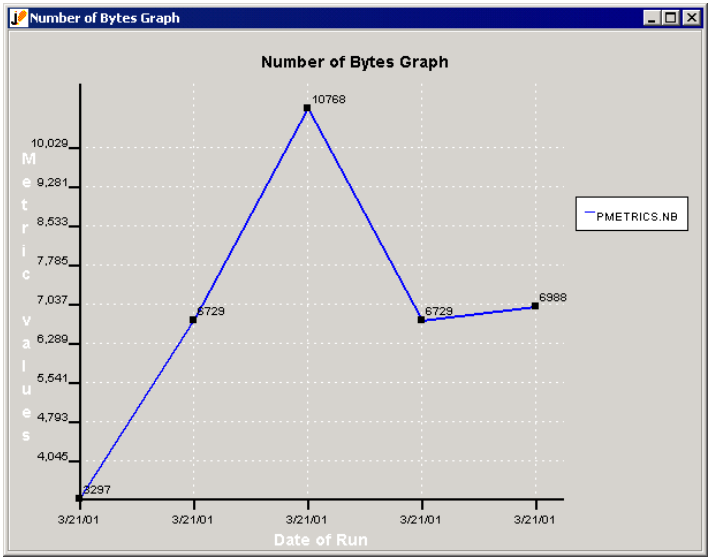
- **Number of bytes:** Total number of bytes of all class files in the project.
- **Number of classes:** Total number of classes in the project.
- **Number of Java source files:** Total number of Java source files in the project.
- **Number of lines:** Total number of lines in the project's classes.
- **Number of packages:** Total number of packages in the project.
- **Number of package-private classes:** Total number of package-private classes in the project.
- **Number of private classes:** Total number of private classes in the project.
- **Number of protected classes:** Total number of "protected" classes in the project.
- **Number of public classes:** Total number of "public" classes in the project.

To track metric information for a specific project:

1. Open the Test History window by clicking the **History** button in the Project Testing UI.
2. Right-click the History for Test node, then choose **Draw Metrics Graph> <Desired Type of Graph>** from the shortcut menu.



Jtest will then create a graph that displays the specified metrics. The graph's X axis contains date information and it's Y axis contain count information. For example, the Number of Lines Graph contains dates on the X axis and the numbers of lines on the Y axis.



Related Topics

“About Static Analysis” on page 69
“Creating Your Own Static Analysis Rules” on page 83
“Viewing Class and Project Metrics” on page 73
“Tracking Metrics Over Time” on page 76
“Customizing Static Analysis” on page 79

Customizing Static Analysis

You can customize Jtest's static analysis feature by:

- Enabling/disabling specific rules.
- Enabling/disabling rule categories
- Customizing built-in rules.

You can also use the RuleWizard feature to create custom coding standards; for more information on this feature, see "Creating Your Own Static Analysis Rules" on page 83.

Enabling/Disabling Specific Rules

One way to control what rules are enforced is to enable/disable individual rules. If you do this, be aware that Jtest looks for violations of a rule only if the rule is enabled and its severity level is enabled.

1. View which rules are currently enabled/disabled.
    a. Open the Global Test Parameters window by clicking the **Global** button.
    b. In the Global Test Parameters window, go to **Static Analysis> Rules**.
    c. Open the **Built-in Rules** node or **User Defined Rules** node.
    d. Open the node for the category of rules that you want to view.
2. Enable/disable rules by right-clicking the appropriate rule and choosing the appropriate command (either **Enable** or **Disable**) from the shortcut menu.

Enabling/Disabling Rule Categories

Each rule has two categories:

- The descriptive category that describes the type of rule (for example, Servlets, Design by Contract, Initialization, User Defined Rules, etc.)
- The severity category that describes the severity of a violation of the rule (violations of coding standards that are most likely to cause an error are level 1; violations of coding standards that are least likely to cause an error are level 5). By default, Jtest reports violations of all coding standards with a severity level of 1, 2, or 3. A rule's severity is indicated by the final character in each rule code, as indicated in the Global Test Parameters window's **Static Analysis> Rules> Built-in Rules** node. For example, PB.SBC has a severity level of 1, and OOP.IIN has a severity level of 5.

To enable/disable all rules in a certain descriptive category:

1. Open the Global Test Parameters window by clicking **Global**.
2. Right-click the **Static Analysis> Rules> <name_of_rule_category**> node. A shortcut menu will open.
3. Choose **Enable All** or **Disable All** from the shortcut menu.

To enable/disable rule all rules in a certain severity categories:

1. Open the Global Test Parameters window by clicking **Global**.
2. Open the **Severity Levels Enabled** node (beneath **Static Analysis> Rules**).
3. Enable/disable categories by right-clicking a category whose status you want to change and choosing **Enable** or **Disable** from the shortcut menu.

You can also enable/disable severity categories at the class and project level. You might want to do this, for example, if you generally want to enforce only the most severe rules, but you want to enforce all rules for a specific class. To enable/disable rules at the class or project level, set the **Severity** flags in Class Test Parameters - Static Analysis (if you are testing a single class) or Project Test Parameters - Static Analysis (if you are testing a set of classes).

Customizing Rules

You can customize three types of rules:

1. Rules saved in <Jtest_install_dir>/brules.
2. Naming conventions.
3. Class metrics rules.

Customizing Rules with RuleWizard

You can customize any rule available in <Jtest_install_dir/brules> with RuleWizard.

To access RuleWizard, right-click the **Rules** button in either Jtest UI, then choose **Launch RuleWizard** from the shortcut menu.

The RuleWizard UI will then open. The RuleWizard User's Guide (accessible by choosing **Help> View** in the RuleWizard UI) contains information on how to create, enforce, and enable/disable custom rules.

You can also edit Naming Convention rules and Metrics rule from the Jtest Global Test Parameters tree.

Customizing Naming Conventions

To edit what naming convention is enforced:

1. Locate the rule that you want to modify in the Global Test parameters tree.
2. Right-click the rule, then choose **Edit Regular Expression** from the shortcut menu.
3. Modify the regular expression in the dialog box that opens.
4. Click **OK** to save your changes.

If you want the naming convention to apply only to certain modifiers (e.g., public, protected, package, private):

1. Locate the rule that you want to modify in the Global Test parameters tree.
2. Right-click the rule, then choose **Edit Optional Modifier** from the shortcut menu.

3. Enter the appropriate modifiers in the dialog box that opens (If you are entering multiple modifiers, use a space character to separate the modifiers' names).
4. Click **OK** to save your changes.

Customizing Class Metrics

To edit the upper and lower thresholds for a class metric:

1. Locate the metric rule that you want to modify in the Global Test parameters tree.
2. Right-click the rule, then choose **Modify Upper Threshold** or **Modify Lower Threshold** from the shortcut menu.
3. Modify the threshold in the dialog box that opens.
4. Click **OK** to save your changes.

Related Topics

"About Static Analysis" on page 69

"Performing Static Analysis" on page 71

"Creating Your Own Static Analysis Rules" on page 83

"Viewing Class and Project Metrics" on page 73

Creating Your Own Static Analysis Rules

You can easily create your own static analysis rules using Jtest's RuleWizard feature. With RuleWizard, you create custom rules by graphically expressing the pattern that you want Jtest to look for when it parses code during static analysis. Rules are created by selecting a main "node," then adding additional elements in a flow-chart-like representation until it fully expresses the pattern that constitutes a violation of the rule. Rules are built by pointing and clicking to add graphical representations of rule elements, then using dialog boxes to make any necessary modifications. No knowledge of the parser is required to write or modify a rule.

To access RuleWizard, right-click the **Rules** button, then choose **Launch RuleWizard** from the shortcut menu that opens.

The RuleWizard UI will then open. The RuleWizard User's Guide (accessible by choosing **Help> View** in the RuleWizard UI) contains information on how to create, enforce, and enable/disable custom rules.

Related Topics

"About Static Analysis" on page 69

"Performing Static Analysis" on page 71

"Customizing Static Analysis" on page 79

"Viewing Class and Project Metrics" on page 73

Static Analysis Suppressions

You can suppress the reporting of rule violation messages by adding a suppression from the Errors Found panel or Results panel.

However, because the rule violation messages will change from class to class, the best way to “suppress” the reporting of warning messages from particular rules or sets of rules is by turning rules (or rule categories) on and off in the manner described in “Customizing Static Analysis” on page 79.

To suppress a particular warning message that appears in the Errors Found panel or Results panel, right-click the message that you want suppress, and choose **Suppress** from the shortcut menu. This action will add that specific warning message to the Static Analysis Suppressions list (in the Class Test Parameters’ **Static Analysis> Suppressed Messages** branch.

Related Topics

“Dynamic Analysis Suppressions” on page 89

“Customizing Static Analysis” on page 79

About Dynamic Analysis

Dynamic analysis involves testing a class with actual inputs. To perform dynamic analysis, Jtest automatically generates, executes and evaluates test cases, and-- where applicable-- executes and evaluates user-defined test cases.

Jtest uses dynamic analysis to perform white-box testing, black-box testing, and regression testing. If your code contains Design by Contract comments, Jtest will use this contract information during dynamic analysis.

Related Topics

“Performing Dynamic Analysis” on page 86

“Customizing Dynamic Analysis” on page 88

“Testing Classes That Reference External Resources” on page 93

“Using Custom Stubs” on page 98

“Setting an Object to a Certain State” on page 106

Performing Dynamic Analysis

Jtest performs dynamic analysis and static analysis each time that you test a class or set of classes.

To perform dynamic analysis:

1. Open the appropriate UI for your test. The Class Testing UI is used to test a single class; the Project Testing UI is used to test a set of classes.
    - 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. Use the **Browse** button to indicate what class or set of classes you want to test.
4. Test the class or project by clicking the **Start** button.
    - If you only want to perform dynamic analysis, right-click the **Start** button, then choose **Dynamic Analysis** from the shortcut menu.

Jtest will then run all requested tests.

Errors found will be reported in the Errors Found Panel (if you tested a single class) or the Results Panel (if you tested a project).

Important: Because Jtest's dynamic analysis tests at the class level, Jtest will only perform dynamic analysis on classes whose .class files are available.

Related Topics

"About Dynamic Analysis" on page 85

"Customizing Dynamic Analysis" on page 88

“Testing Classes That Reference External Resources” on page 93
“Using Custom Stubs” on page 98
“Setting an Object to a Certain State” on page 106
“Jtest Tutorials” on page 268

Customizing Dynamic Analysis

Dynamic analysis can be customized by suppressing dynamic analysis error messages, and by modifying Class, Project, and Global Test Parameters.

- The parameters under **Test Case Generation** control the generation of both the automatic test cases (the ones that Jtest generates automatically) and the user-defined test cases.
- The parameters under **Test Case Execution** control the execution of all the test cases.
- The parameters under **Test Case Evaluation** control the evaluation of all the test cases. Note that the evaluation is performed on both the automatic and user-defined test cases. For example, if **Perform Automatic Regression Testing** is selected, automatic regression testing is performed for both the automatic and the user-defined test cases.

Related Topics

“About Dynamic Analysis” on page 85

“Performing Dynamic Analysis” on page 86

“Testing Classes That Reference External Resources” on page 93

“Using Custom Stubs” on page 98

“Setting an Object to a Certain State” on page 106

Dynamic Analysis Suppressions

In general, the best way to prevent Jtest from reporting exceptions that are not relevant to the class under test is to document the class's permissible inputs and/or expected exceptions using Design by Contract tags. This way, the class's implicit contracts are documented in the code itself. For information about using these tags to suppress exceptions, see "Customizing White-Box Testing" on page 112

Another preferred way to stop Jtest from displaying certain uncaught runtime exceptions, specification errors, and regression errors is to right-click the appropriate message in the Errors Found panel of the Class Testing UI or in the Results panel of the Project Testing UI, then choose one of the following commands from the shortcut menu:

- **Not an Error:** Choose this command to indicate that a reported problem is not an error, but rather is the class's correct behavior (for that input).
- **Ignore:** Choose this command to tell Jtest to ignore the problem reported for this input. If this is selected, the outcome will not be used for comparisons when searching for specification or regression errors. Also, no uncaught runtime exceptions will be reported for this input.

You can also suppress the reporting of uncaught runtime exceptions in two ways:

- Adding a suppression from the Errors Found panel or Results panel.
- Adding a suppression directly to the Dynamic Analysis Suppressions Table.

To suppress a particular exception that appears in the Errors Found panel or Results panel, right-click the exception that you want suppress, and choose **Suppress** from the shortcut menu. This action will add that specific exception to the Dynamic Analysis Suppressions Table.

You can enable or disable checking for certain types of exceptions in the **Pre-filtering Suppressions Categories** node (located under **Dynamic**

Analysis> Test Case Execution) of the Global, Project, or Class Test Parameters.

The Dynamic Analysis Suppressions Table lets you create new suppression categories for uncaught runtime exceptions. You can use this option to suppress the reporting of exceptions by class, method, and exception type.

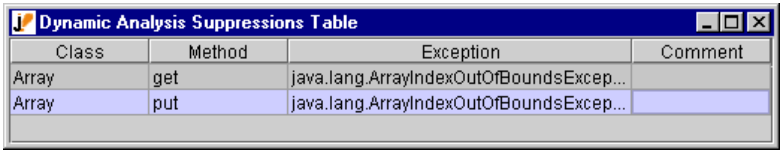
Opening the Suppressions Table

To reach the Suppressions Table, double-click the **Suppressions Table** node in the **Dynamic Analysis** branch of the Global Test Parameters tree.

Adding a Suppression to the Dynamic Analysis Suppressions Table

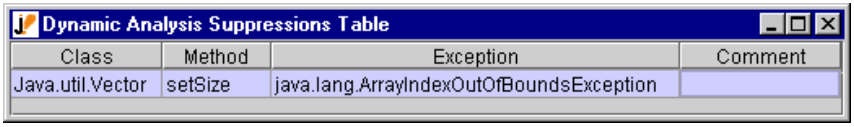
1. Right-click any area of the table. A shortcut menu will open.
2. Choose **Add New Suppression** from the shortcut menu. An empty table entry will open.
3. In the empty table entry, enter the exception, the method that throws it, or the class that declares the method that you want to suppress.

Important: Make sure that you type all values exactly as they appear in the Jtest UI.



Suppressing Specific Exceptions by Class, Method, and Exception Type

To suppress specific exceptions by class, method, and exception type, enter the information in the appropriate fields. For example, to suppress the ArrayIndexOutOfBoundsException from the 'setSize' method of java.util.Vector, use:



Suppressing Exceptions by Class

To suppress exceptions by class, enter the classname in the **Class** field by double clicking that field and then entering the fully qualified class-name.

Use the '*' (asterisk) symbol to match any letter. For example, to suppress all exceptions from the class java.util.Vector, use:

```
java.util.Vector | * | *
```

Suppressing Exceptions by Method

To suppress exceptions by method, enter the method name in the **Method** field by double-clicking that field then entering the method name. To suppress exceptions from a single method belonging to a set of over-loaded methods, specify the method by including its signature enclosed in parentheses. Method signatures should follow the JNI specification from the JDK (minus the return type). For example, to suppress exceptions from the following method

```
(int n, String s, int []ia)
```

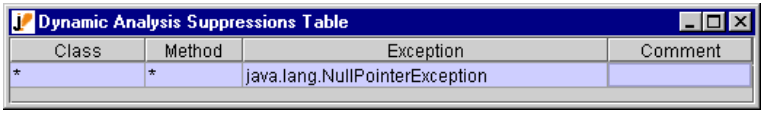
use the following as the method signature:

```
(ILJava/lang/String;[I)
```

Suppressing Exceptions by Exception Type

To suppress exceptions by exception type, enter the exception type in the **Exception** field by double clicking that field and then entering the exception type.

Use the '*' (asterisk) symbol to match any letter. For example, to suppress all NullPointerExceptions use:



Removing a Suppression

1. Right-click the table entry. A shortcut menu will open.
2. Choose **Delete** from the shortcut menu.

Note: To delete a list of suppressions, right-click drag from the first suppression to the final one, then release the mouse button. A shortcut menu will open. Choose **Delete** from the shortcut menu.

Viewing the Results

Suppressions in this list are applied when the results are displayed. Results are displayed after you click the **Start** button, generate the report file, or view results in the Results UI.

Related Topics

“Customizing White-Box Testing” on page 112

“Static Analysis Suppressions” on page 84

Testing Classes That Reference External Resources

Jtest's multiple stub options let you automatically and precisely test classes that reference external resources.

Stubs are basically replacements for references to external methods. For example, you could use stubs to specify that when the method "stream.readInt()" is invoked, Jtest should use the value 3 instead of actually invoking the readInt method.

Stubs are mainly used:

- To isolate a class and test it independently of other classes, or
- To test a class before the external classes it uses are available.

If you are using automatically-generated test cases to test classes that reference external resources, you can:

- Have Jtest automatically generate stubs,
- Enter your own stubs, or
- Have Jtest call the actual external method.

Jtest does not automatically generate stubs for user-defined test cases. If you are using user-defined test cases to test classes that reference external resources, you can:

- Enter your own stubs, or
- Have Jtest call the actual external method.

When you perform regression testing on classes that reference external resources, Jtest will automatically use the stub types (if any) that were used during the previous test run(s).

Defining Which Classes are “External”

You can indicate which classes are external by defining the “Tested Set”.

The Tested Set is the set of classes and methods included in the current test. Any class outside of the Tested Set is considered external. When a class or method in the Tested Set references a class or method that is inside that Tested Set, the actual class or method is accessed. When a class or method in the Tested Set references a class or method that is outside that Tested Set, stubs are called.

The class under test and its inner classes are always included in the Tested Set. When Jtest creates a new project, its default Tested Set is comprised of all of the classes in the package and subpackages of each class under test.

The Tested Set will also include additional classes that match the prefixes specified in the Tested Set list.

To open the dialog box that lets you specify what other classes are included in the Tested Set, open the appropriate parameters window, then:

1. Open **Dynamic Analysis> Test Case Execution> Stubs.**
2. Double-click the **Tested Set Includes** node.
3. To add a class name prefix, enter or browse to it, then click **Add**. To remove a class name prefix, select it, then click **Delete**. You can use a specific class name prefix as well as any of the following tokens:
    - $PACKAGE: This token is replaced by the name of the package under test.
    - $PARENT: This token is replaced by the parent parameter value.
    - <unnamed>: This token refers to the unnamed package.
4. Click **OK** to close the Tested Set dialog box.

Using Your Own Stubs

When you configure Jtest to use your own stubs, you have complete control over what values or exceptions an external method returns to the class under test-- without having to have the actual external method completed and/or available. You can enter your own stubs for both automatically-generated and user-defined test cases.

If a stub is not defined for an external method or if no options in the appropriate test parameter's **Dynamic Analysis> Test Case Execution> Stubs> Options for Automatic Test Cases/ Options for User Defined Test Cases** options are enabled, Jtest will call the actual external method.

For more information about entering your own stubs, see "Using Custom Stubs" on page 98.

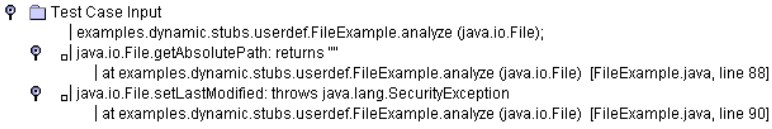
Automatically-Generated Stubs

When performing white-box testing on classes that reference external resources (such as external files, databases, Enterprise Java Beans (EJB) and CORBA), Jtest automatically generates stubs for the resources and executes the stubs to get input for the call to the external resources; like Jtest's other automatically-generated inputs, these inputs are designed to provide maximum coverage of the class. When designing these inputs, Jtest assumes that a call to an external resource can return any input compatible with its return type.

These inputs are created as white-box stubs, and these stubs are used for both white-box testing and regression testing. When these stubs are used for white-box testing, Jtest executes the stubs and reports errors if any uncaught runtime exceptions occur from the stubs' input. When these stubs are used for regression testing, Jtest executes the stubs and reports errors if class modifications cause previously tested input to produce output other than the known or previous value.

You can view the stubs that Jtest automatically generated and executed in the **Automatic Test Cases> Method Name> Test Case> Test Case Input** branch of the View Test Cases tree. Automatically-generated stubs will be marked with a small, empty box. Each stub branch displays the method invoked as well as the value or exceptions returned by the stub.

Expand the stub's branch to see the stack trace where the invocation occurred.



If the stub's values resulted in an error, the above information will also be displayed in the Errors Found Panel (if you are testing a class) or the Results panel (if you are testing a project).

Currently, Jtest can generate inputs for the following external resources:

- java.io
- java.net
- java.sql

In addition, Jtest offers preliminary support for CORBA and Enterprise Java Beans (see below for details).

Note: This support for classes that reference external resources is preliminary and we welcome any suggestions you have on improving it.

CORBA

When you perform white-box testing on classes that call CORBA objects, Jtest automatically generates and executes white-box stubs for the Object Request Broker and for other CORBA objects referenced by the class under test. For example, if a client or CORBA class references another CORBA object, Jtest will assume that method calls to the other CORBA object can return any value compatible with its return type, generate and execute white-box stubs that contain appropriate input, and report any uncaught runtime exceptions that result from this input.

When you test a set of classes using the Project UI, Jtest skips automatically generated classes such as helper classes, client stub classes, server skeletons, etc..

Enterprise Java Beans

When you perform white-box testing on EJB classes, Jtest assumes that the beans referenced by the bean under test can return any value that is compatible with the return type for that method. This tests that the EJB bean class will behave correctly regardless of the other bean or beans' behavior or return values.

Before you test any business method in the EJB class, Jtest will invoke the bean initialization routines and provide a dummy container context. Previous Jtest versions called the class's constructor and invoked the business methods. If you prefer to have Jtest test business methods using this older technique, set the following environment value:

For Windows: `set JTEST_OPT_SKIP_CREATE=ON`

For UNIX: `setenv JTEST_OPT_SKIP_CREATE ON`

If you are using the Weblogic server, Jtest skips the classes that the EJB vendor generated automatically. If you are using a different server and would like this feature, please contact ParaSoft technical support.

Related Topics

"About Dynamic Analysis" on page 85

"Performing Dynamic Analysis" on page 86

"Customizing Dynamic Analysis" on page 88

"Using Custom Stubs" on page 98

"Setting an Object to a Certain State" on page 106

Using Custom Stubs

Stubs are basically replacements for references to external methods. For example, you could use stubs to specify that when the method "stream.readInt()" is invoked, Jtest should use the value “3” instead of actually invoking the readInt method.

Stubs are mainly used:

- To isolate a class and test it independently of other classes, or
- To test a class before the external classes it uses are available.

You can enter your own stubs for both automatic and user-defined test cases. When you configure Jtest to use your own stubs, you have complete control over what values or exceptions an external method returns to the class under test-- without having to have the actual external method completed and/or available.

There are 5 basic steps involved in creating and using user defined stubs:

1. Create the custom stub.
2. Enable the custom stub.
3. Indicate the stub’s location.
4. Test the class in the normal manner.
5. View the stubs.

Creating a Custom Stub

Standard Procedure

The first step in using user defined stubs is creating a Stubs Class. If you create a Stubs Class named “(name_of_class_under_test_Stubs)” and save it in the same directory as the class under test, you will not have to indicate the stub’s location. You can use a class with a different name or location as long as you indicate the stub’s location (as described in Indicating the Stub’s Location below).

The main way to specify stubs is to create a Stubs Class: a class that contains one or more “stubs()” methods which define what (if any) return val-

ues or exceptions should be used for a certain input. Stubs Classes extend “jtest.Stubs”. For information about jtest.Stubs, see the Jtest API javadoc (you can access this documentation by choosing **Help> Jtest API**).

Each Stubs Class should implement a method of the form:

```
public static Object stubs (
   Method method // external method being invoked
   , Object _this // "this" if instance method
   , Object[] args // arguments to the method
                   // invocation
   , Method caller_method // method calling "method"
   , boolean executing_automatic  // true if executing
                                  //automatic Test Case
);
```

Important: Only the first parameter is required; all others are optional. For example, you could define a “stubs() method” of the form “Object stubs (Method method)”.

Whenever the class under test invokes a method “method” external to the class, Jtest will call the “stubs()” method. The "stubs()" method should declare what (if any) return values or exceptions should be returned for certain inputs. Use the following table to determine what type of return values or exceptions should be used for each possible stub type:

If you want the external method to return this . . .	**Have the “stubs()” method return this . . .**
no stub	NO_STUBS_GENERATED
void	VOID
a value	The value. If the value is a primitive type, it should be wrapped in an Object. For example, if “stubs()” decides that a given external call should return the integer 3, “stubs()” should return “new Integer (3)”.

an exception	The exception that you want the stub to throw. For example, you could use something like: `throw new IllegalArgumentException (“time is:” + time);`

Important: The “stubs()” method can only return an Object. To specify a return value of a primitive type, you need to wrap that type in an Object. For example, if you wanted to specify that the method

```
int userMethod()
```

returned 3, you should have the "stubs ()" method return

```
return new Integer (3)
```

To define stubs for constructor invocations, define a “stubs()” method whose first parameter is a constructor (instead of a method).

If some “stubs()” method is not defined, no stubs will be used for those members (method or constructor).

Stub Objects

Stub objects are very useful when writing user defined stubs. A stub object is similar to any other object, with the following differences:

- The stub object can be an instance of an interface. For example, the following creates an instance of "Enumeration":

  ```
  Enumeration enum = makeStubObject (Enumera-
  tion.class);
  ```

- Any method invocation or field reference is a stub even if no stub has been defined for it. If no stub has been defined, Jtest will use a default stub returning the default initialization value for the method return type or field type (for example, "null" for Object, "0.0d" for double,etc.).

You need to use stub objects if you want to test classes that use interfaces for which an implementation has not yet been written. They can be used whenever an object of the interface class needs to be created.

Stub objects can also be used whenever you want to create an object of a given type without having to call a specific constructor. For example, instead of using "new java.io.FileInputStream ("what to put here?)", you could use: "(FileInputStream) JT.makeStubObject (java.io.FileInput-Stream.class)"; this creates a FileInputStream object.

Defining Stubs in Test Classes

If you are using a Test Class, you can define specific stubs for each test method by defining a "stubs()" methods within the Test Class. For example, to specify the stubs for a test case defined by a "testXYZ" method, define a method of the form:

```
Object stubsXYZ (Method method, ...);
```

in that Test Class.

If a Test Class does not define a "stubs ()" method, or if it does not return any stubs, Jtest will apply the Class and Project Test Parameters "stubs()" methods.

For more information on Test Classes, see “Adding Test Classes” on page 125.

Defining Stubs at the Project Level

If more than one of the classes in your project uses the same "stubs ()" method, you should create a project-level Stubs Class that contains the return values for that method. Project-level stubs should be created like class-level stubs. The only differences between project-level stubs and class-level stubs are:

- Class-level stubs contain stubs specific to a class, whereas project-level stubs contain stubs that can be shared by multiple classes.
- You indicate the location of class-level stubs in the Class Test Parameters **Dynamic Analysis> Test Case Execution> Stubs> Stubs Class** branch, but you indicate the location of project-level stubs in the Project Test Parameters **Dynamic Analysis> Test Case Execution> Stubs> Stubs Class** branch.

When Jtest tests a class in the project, it will first apply the Stubs Class indicated in the Class Test parameters. If no stub is generated at the class level, Jtest will apply the Stubs Class indicated at the project level.

Enabling User Defined Stubs

Jtest will not use your user defined stubs unless it is configured to do so. By default, user defined stubs are enabled for user defined test cases, but disabled for automatically-generated test cases.

You can enable user defined stubs at the global, project, or class level. To do so, open the appropriate parameters window, then:

1. Open **Dynamic Analysis> Test Case Execution> Stubs.**
2. Open either **Options for Automatic Test Cases** or **Options for User Defined Test Cases**.
3. Enable the **Use User Defined Stubs** option.

Indicating the Stub's Location

By default, when Jtest detects that the class under test references an external method, it searches for and uses Stubs Classes that are:

- In the same directory as the class under test, and
- Named <name_of_class_under_test>Stubs (for example, fooStubs.class).

If you do not change the default setting (Class Test Parameters' **Dynamic Analysis> Test Case Execution> Stubs> Stubs Class** value set to "$DEFAULT"), your Stubs Classes are named correctly, and your Stubs Classes are located in the same directory as the class under test, you do not need to indicate the stub's location.

If you need to indicate the Stubs Class location for a specific class, right-click the Class Test Parameters' **Dynamic Analysis> Test Case Execution> Stubs> Stubs Class** node, choose **Edit**, then enter the location of the Stubs Class.

If you want to indicate the Stubs Class's location at the project level, right-click the Project Test Parameters' **Dynamic Analysis> Test Case**

Execution> Stubs> Stubs Class node, choose **Edit**, then enter the location of the Stubs Class.

Important: If you specify a Stubs Class at the project level, Jtest will first apply the Stubs Class indicated in the Class Test parameters. If no stub is generated at the class level, Jtest will apply the Stubs Class indicated at the project level. Jtest will not automatically search for Stubs Classes at the project level.

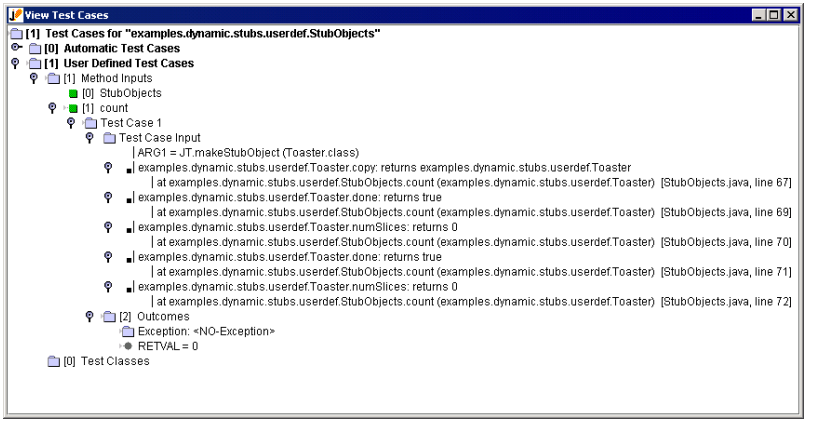
If you specified your stub via a Test Class, you do not need to indicate the location of a Stubs Class.

Running the Test

If you have performed the above steps, Jtest will automatically use your stubs when you test the class or project in the normal manner.

Viewing the Stubs

After you test a class using a user defined stub, you can view the stubs in the **User Defined Test Cases> Method Name> Test Case> Test Case Input** branch of the View Test Cases tree. User defined stubs will be marked with a small black box. Each stub branch displays the method invoked as well as the value or exceptions returned by the stub. Expand the stub's branch to see the stack trace where the invocation occurred.



If the stub's values resulted in an error, the above information will also be displayed in the Errors Found Panel (if you are testing a class) or the Results panel (if you are testing a project).

Summary

If you want Jtest to use user defined stubs:

1. Create a Stubs Class whose "stubs ()" methods indicate what (if any) values or exceptions you want the stub to return.
2. Enable user defined stubs in the appropriate test parameters window.
3. If your Stubs Class is not named (name_of_class_under_test)Stubs and is not in the same directory as the class under test (or, if your Stubs Class should be applied at the project level), indicate the stub's location.
4. Test the class as normal.

Related Topics

"About Dynamic Analysis" on page 85.

“Performing Dynamic Analysis” on page 86.
“Customizing Dynamic Analysis” on page 88.
“Testing Classes That Reference External Resources” on page 93.
“Setting an Object to a Certain State” on page 106.

Setting an Object to a Certain State

In some cases, you may want to set up an initial state prior to testing a class. For example, suppose that a class is used as a global object containing static member variables accessible by any other project within the application. When Jtest tests an object that uses this static member variable, a NullPointer Exception will result because the variable has not been set. This problem can be solved by giving Jtest static initialization code.

About Static Initialization Code

All initialization code will be executed before any test case is executed, and can be used to setup and initialize the class if needed.

You can add static initialization code at the global, project, or class level. Initialization code set in the Global Test Parameters will be executed for all classes that Jtest tests. Initialization code set in the Project Test Parameters will be executed for all classes in the project. Initialization code set in the Class Test Parameters will be executed only for the class whose parameters you are editing.

Initialization code is executed in the following order:

1. Static Global Initialization code
2. Static Project Initialization code
3. Static Class Initialization code

For an example that uses initialization code, see <jtest_install_dir>/examples/dynamic/common/ClassInit.

Note: Initialization code can only be used to invoke static methods.

Adding Static Initialization Code

To add static initialization code:

1. Open the Test Parameters window for the level at which you want to add initialization code.
    - To add Global Initialization code, click **Global** (in either UI).
    - To add Project Initialization code, click **Project** (in the Project Testing UI).
    - To add Class Initialization code, click **Class** (in the Class Testing UI).
2. In the Test Parameters window, open **Dynamic Analysis> Test Case Generation> Common**.
3. Double-click the **Static Global Initialization**, **Static Project Initialization**, or **Static Class Initialization** node. A Static Initialization window will open.
4. Enter the initialization code in the Static Initialization window, or import the code by choosing **Options> Import**.
5. To save the modification, choose **Options> Save**.
6. To exit the Static Initialization window, choose **Options> Quit**.

Related Topics

“About Dynamic Analysis” on page 85

“Performing Dynamic Analysis” on page 86

“Customizing Dynamic Analysis” on page 88

“Testing Classes That Reference External Resources” on page 93

“Using Custom Stubs” on page 98

About White-Box Testing

White-box testing checks that the class is structurally sound. It doesn't test that the class behaves according to the specification, but instead ensures that the class doesn't crash and that it behaves correctly when passed unexpected input. White-box testing involves looking at the class code and trying to find out if there are any possible class usages that will make the class crash (in Java this is equivalent to throwing an uncaught runtime exception).

Jtest uses unique technology to completely automate the white-box testing process. Jtest examines the internal structure of each class under test, automatically designs and executes test cases designed to fully test the class's construction, then determines whether each test case's inputs would produce an uncaught runtime exception. For each uncaught runtime exception that is detected, Jtest reports an error and provides the stack trace as well as the calling sequence that led to the problem. \

Jtest can perform white-box testing on any Java class or component, including classes that reference external resources (such as external files, databases, Enterprise JavaBeans™ [EJB] and CORBA). If you are performing white-box testing on classes that reference external resources, Jtest will automatically generate the necessary stubs, or give you the option of calling the actual external method or entering your own stubs. For classes using CORBA, Jtest provides stubs for the Object Request Broker and other objects referenced by the class. For classes using EJB, Jtest invokes bean initialization routines and provides a simulated container context, then performs white-box testing to make sure that the bean class will always behave correctly.

If you find that certain exceptions reported are not relevant to the project at hand, you can easily tailor Jtest's error reports to your needs. If you document valid exceptions in the code using a special `@exception` comment tag, Jtest will suppress any occurrence of that particular exception. If you use the `@pre` comment tag to document the permissible range for valid method inputs, Jtest will suppress errors found for inputs that fall outside of that range. You can also suppress exceptions using shortcut menus or the suppression panel.

Related Topics

“Performing White-Box Testing” on page 110

“Customizing White-Box Testing” on page 112

Performing White-Box Testing

Jtest performs white-box testing, along with all other appropriate types of testing, each time that you test a class or set of classes.

To perform white-box testing:

1. Open the appropriate UI for your test. The Class Testing UI is used to test a single class; the Project Testing UI is used to test a set of classes.
    - 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. Use the **Browse** button to indicate what class or set of classes you want to test.
4. Test the class or project by clicking the **Start** button.
    - If you only want to perform dynamic analysis, right-click the **Start** button, then choose **Dynamic Analysis** from the shortcut menu.
    - If you only want to execute automatically-generated test cases, right-click the **Start** button, then choose **Dynamic Analysis> Automatic** from the shortcut menu.

Jtest will then run all requested tests.

Uncaught runtime exceptions found will be reported in the **Uncaught Runtime Exceptions** branch of the Errors Found Panel (if you tested a single class) or the Results Panel (if you tested a project).

Suppressed Exceptions

The following types of exceptions are suppressed by default:

- Exceptions in Throws Clause
- DirectIllegalArgumentExceptions
- Explicitly Thrown Exceptions
- Exceptions Caught By Empty Catch
- Direct NullPointerExceptions

You can enable the reporting of any of these exception types by modifying settings in any parameter tree's **Dynamic Analysis> Test Case Execution> Pre-Filtering Categories** branch.

All exceptions found-- including suppressed exceptions-- are displayed in the View Test Cases window. To see the reason why one of the exceptions listed here was suppressed, right-click that exception's node (the node with the lightning bolt icon) in the View Test Cases window, then choose **Why Suppressed?** from the shortcut menu.

For information on determining what types of exceptions you want suppressed, see "Customizing White-Box Testing" on page 112 and "Dynamic Analysis Suppressions" on page 89.

Related Topics

"About White-Box Testing" on page 108

"Customizing White-Box Testing" on page 112

"Jtest Tutorials" on page 268

Customizing White-Box Testing

You use two Design by Contract tags to customize Jtest so that it automatically suppresses uncaught runtime exceptions that you do not expect to occur or that you are not concerned with.

To have Jtest suppress errors for inputs that you do not expect to occur, use the @pre tag to specify what inputs are permissible. If you use the @pre tag to indicate valid method inputs, then use Jcontract to check Design by Contract contracts at runtime, you will automatically be alerted to instances where the system passes this method these unexpected inputs.

To have Jtest suppress expected exceptions, use the @exception tag to specify what exceptions you want Jtest to ignore.

For details on using these two tags, see “About Design by Contract” on page 137 and “The Design by Contract Specification Language” on page 141.

Related Topics

“About White-Box Testing” on page 108

“Performing White-Box Testing” on page 110

“Dynamic Analysis Suppressions” on page 89

“Using Design by Contract With Jtest” on page 133

“The Design by Contract Specification Language” on page 141

“Testing A Class - Two Simple Examples” on page 23

About Black-Box Testing

Black-box (functionality) testing checks a class's functionality by determining whether or not the class's public interface performs according to specification. This type of testing is performed without paying attention to implementation details.

If your class contains Design by Contract-format specification information, Jtest completely automates the black-box testing process. If not, Jtest makes the black-box testing process significantly easier and more effective than it would be if you were creating test cases by hand.

Jtest reads specification information built into the class with the DbC language, then automatically develops test cases based on this specification. Jtest designs its black-box test cases as follows:

- If the code has postconditions, Jtest creates test cases that verify whether the code satisfies those conditions.
- If the code has assertions, Jtest creates test cases that try to make the assertions fail.
- If the code has invariant conditions (conditions that apply to all of a class's methods), Jtest creates test cases that try to make the invariant conditions fail.
- If the code has preconditions, Jtest tries to find inputs that force all of the paths in the preconditions.
- If the method under test calls other methods that have specified preconditions, Jtest determines if the method under test can pass non-permissible values to the other methods.

Jtest also helps you create black-box test cases if you do not use Design by Contract. You can use Jtest's automatically-generated set of test cases as the foundation for your black-box test suite, then extend it by adding your own test cases.

Test cases can be added in a variety of ways; for example, test cases can be introduced by adding:

- Method inputs directly to a tree node representing each method argument.

- Constants and methods to global or local repositories, then adding them to any method argument.
- JUnit-format Test Classes for test cases that are too complex or difficult to be added as method inputs.

If a class references external resources, you can enter your own stubs or have Jtest call the actual external method.

When the test is run, Jtest uses any available stubs, automatically executes the inputs, and displays the outcomes for those inputs in a simple tree representation. You can then view the outcomes and verify them with the click of a button. Jtest automatically notifies you when specification and regression testing errors occur on subsequent tests of this class.

Related Topics

“Performing Black-Box Testing” on page 115

“Adding Method Inputs” on page 119

“Adding Test Classes” on page 125

“Specifying Imports” on page 132

Performing Black-Box Testing

Jtest performs black-box testing, along with all other appropriate types of testing, each time that you test a class or set of classes.

Jtest will automatically create and execute test cases that verify code functionality when specification information is incorporated into the code using the Design by Contract language. For information on adding Design by Contract-format contracts to your code, see “Using Design by Contract With Jtest” on page 133. and “The Design by Contract Specification Language” on page 141..

Jtest will also check functionality using any test cases you have added as well as any automatically-generated test cases whose outputs you have validated.

Important: In order to perform black-box testing, Jtest needs to know the location of your JDK. Jtest determines this location automatically. For information on changing the JDK used, see “JDK Requirement” on page 15.

To perform black-box testing:

1. Open the appropriate UI for your test. The Class Testing UI is used to test a single class; the Project Testing UI is used to test a set of classes.
    - 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. Use the **Browse** button to indicate what class or set of classes you want to test.
4. (Optional) Add test cases by adding method inputs and/or Test Classes.
    - For information on adding method inputs, see "Adding Method Inputs" on page 119..
    - For information on adding Test Classes, see "Adding Test Classes" on page 125..
5. Test the class or project by clicking the **Start** button.
    - If you only want to perform dynamic analysis, right-click the **Start** button, then choose **Dynamic Analysis** from the shortcut menu.
    - If you only want to execute automatically-generated test cases, right-click the **Start** button, then choose **Dynamic Analysis> Automatic** from the shortcut menu.
    - If you only want to execute user-defined test cases, right-click the **Start** button, then choose **Dynamic Analysis> User Defined** from the shortcut menu.

Jtest will then run all requested tests.

If the classes under test contain Design by Contract specification information, any functionality problems found will be reported in the **Design by Contract Violations** branch of the Errors Found Panel (if you tested a single class) or the Results Panel (if you tested a project).

If you added user-defined test cases, you should evaluate the outcomes for all tested classes and specify the correct output values for test cases that failed.

To evaluate test case outcomes for a class:

1. Review the class's test case outcomes in the View Test Cases window.
    - To open this window from the Class Testing UI, click the **View Test Cases** button.
    - To open this window from the Project Testing UI's Results panel, right-click the **[Class Name]** node, then choose **View Test Cases** from the shortcut menu.
2. In the View Test Cases window, expand the test case tree so that the inputs and outcomes for the test cases you are evaluating are visible.
3. Indicate whether or not the outcome for each test case is correct by right-clicking the appropriate outcome, then choosing the appropriate option.
    - Choose **Mark as Correct** if the listed outcome is the expected outcome.
    - Choose **Mark as Incorrect** if the listed outcome is not the expected outcome.
    - Choose **Mark as Unknown** if you don't know how the listed outcome compares to the expected outcome.
    - Choose **Mark as Ignore** if you want Jtest to ignore the listed outcome.
    - To choose the same option for all of a test case's outcomes, right-click the test case's **Outcomes** leaf, then choose the appropriate **Set All to...** command from the shortcut menu.
4. If any outcome was incorrect, enter the correct value by:
    a. Opening the Class Test Parameters window.

b. Opening that test case's branch in **Dynamic Analysis> Test Case Evaluation> Specification and Regression Testing**.

c. Right-clicking the outcome, choosing **Edit** from the short-cut menu, then entering the correct value in the text field that opens.

Now every time Jtest is run on that class, it will check whether or not the correct outcomes are produced.

Any problems found using these test cases will be reported in the **Specification and Regression Errors** branch of the Errors Found Panel (if you tested a single class) or the Results Panel (if you tested a project).

Related Topics

"About Black-Box Testing" on page 113.

"Adding Method Inputs" on page 119.

"Adding Test Classes" on page 125.

"Specifying Imports" on page 132.

"Using Design by Contract With Jtest" on page 133.

"The Design by Contract Specification Language" on page 141.

"Testing A Class - Two Simple Examples" on page 23.

"Jtest Tutorials" on page 268.

Adding Method Inputs

Jtest allows you to add both primitive and complex method inputs. The procedure for adding inputs varies depending on the type of input that you want to add.

Important: Method inputs are added to arguments in the Class Test Parameters window.

If you are in the Class Testing UI, you can open this window by clicking the **Class** button.

If you are in the Project Testing UI, you can open this window by performing the following steps:

1. Click **Project**.
2. Right-click the **Classes in Project> <name of class to which you want to add inputs>** branch, then choose **Edit Class Test Parameters** from the shortcut menu.

Adding Primitive Inputs

There are two ways to define your own primitive inputs for methods under test:

- Using the Method's Class Test Parameters Tree Node
- Using the Repository

Using the Method's Class Test Parameters Tree Node

To add primitive inputs directly to the method's Class Test Parameters Tree node:

1. Open the Class Test Parameters window.
2. In the Class Test Parameters window, go to **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs** to view a list of all methods in the class.

3. Open the node associated with the method whose inputs you want to define.
4. Right-click the argument that you want to define an input for, then choose **Add Input Value** from the shortcut menu.
5. In the text field of the box that opens, type the input that you want to use, then press **Enter** to save this value.

Using the Repository

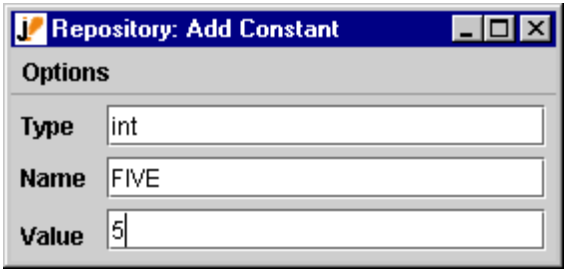
To add a primitive inputs using the repository, you must complete two main tasks:

1. Add constants to the inputs repository.
2. Add the appropriate repository inputs to the appropriate method argument.

Adding Constants to an Inputs Repository

To add a constant to an Inputs Repository:

1. Open the Class Test Parameters or Global Test Parameters window.
2. In the Test Parameters window, go to **Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.
3. Right-click **Inputs Repository**, then choose **Add Constant** from the shortcut menu. The Add Constant window will open.
4. In the Add Constant window, enter the type of the constant (e.g. int) in the **Type** field, the name of the constant (e.g. FIVE) in the **Name** field, and the value of the constant (any valid Java expression-- e.g., 5) in the **Value** field.



5. Choose **Options> Save**.
6. Choose **Options> Quit**.

Using Repository Inputs

When you want to add repository inputs to an argument:

1. In the Class Test Parameters window, right-click the node associated with the argument that you want to add an input value to (this node is at **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs> <Method Name**>). A shortcut menu will open.
2. From the shortcut menu, choose either **Add From Local Repository** (if the input is in the local repository), or **Add From Global Repository** (if the input is in the global repository), then choose the desired input.

Adding Non-Primitive Inputs

There are two ways to add non-primitive (object-type) inputs to a method:

- Using a .java Class File
- Using the Repository

Using a .java Class File

The objects to be used for the user-defined test cases can be defined in any .java class. Jtest can use those inputs as long as:

- The class that is defining the input object is in the classpath.
- You import each input class using the **Dynamic Analysis> Test Case Generation> Common> Imports** node of the Class Test Parameters tree.

For an example of how to add inputs using .java class files, see <jtest_install_dir>/examples/blackbox/inputs/README.

Using the Repository

To add a non-primitive input using the repository, you must complete two main tasks:

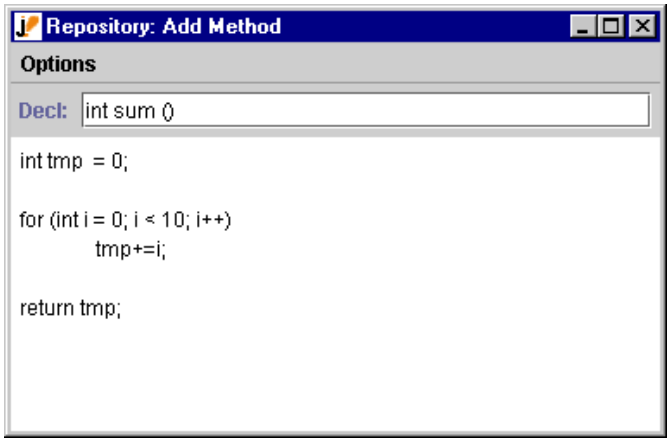
1. Define a method that will instantiate and set up the desired object input, then add it to an Inputs Repository.
2. Add the appropriate repository input to the appropriate method argument.

Adding Methods to an Inputs Repository

To add a method to an Inputs Repository:

1. Open the Class Test Parameters or Global Test Parameters window.
2. In the Test Parameters window, go to **Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.
3. Right-click **Inputs Repository**, then choose **Add Method** from the shortcut menu. The Add Method window will open.

In that window, define a method that creates and returns the desired input values.



4. Enter the method declaration (e.g. int sum()) in the **Decl** field.
5. (Optional) If you want to check the method, choose **Save & Check** from the **Options** menu.
6. Choose **Options> Save**.
7. Choose **Options> Quit**.

Using Repository Inputs

When you want to add repository inputs to an argument:

1. In the Class Test Parameters window, right-click the node associated with the argument that you want to add an input value to (this node is at **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs> <Method Name**>). A shortcut menu will open.
2. From the shortcut menu, choose either **Add From Local Repository** (if the input is in the local repository), or **Add From Global Repository** (if the input is in the global repository), then choose the desired input.

Related Topics

“About Black-Box Testing” on page 113

“Performing Black-Box Testing” on page 115

“Adding Test Classes” on page 125

“Specifying Imports” on page 132

“Jtest Tutorials” on page 268

Adding Test Classes

Test Classes let you add test cases that are too complex or difficult to be added as method inputs. For example, you might want to use Test Classes if you want to:

- Use objects as inputs for static and instance methods.
- Test a calling sequence and check the state of the object using asserts.
- Create complicated test cases that depend upon a specific calling sequence.
- Validate the state of an object.

A Test Class is a class that extends jtest.TestClass and is used to specify test cases that Jtest should use to test the class. For information about jtest.TestClass, see the Jtest API javadoc (you can access this documentation by choosing **Help> Jtest API**). The jtest.TestClass file is in the jtest.zip file located in <jtest_install_dir>/classes.

Test Classes give you an easy way to write complicated test cases and to verify the state of the objects under test. You can write your own Test Classes using any Java development environment, or you can integrate JUnit classes into your Jtest tests.

To add Test Classes, you need to:

1. Set you environment to use Test Classes.
2. Create a Test Class (or locate a JUnit Test Class)
3. Load the Test Class (this is not necessary if you have named and saved the Test Class in the conventional manner).

These steps are described in detail below.

Setting Your Environment to Use Test Classes

In order to use the Test Classes feature, you need to add the jtest.zip file that contains jtest.TestClass to your CLASSPATH. This zip file is located in <jtest_install_dir>/classes/jtest.zip

You can have Jtest automatically set your CLASSPATH by opening a command prompt, changing directories to the Jtest installation directory, then entering the appropriate command(s):

- If you are using Windows, enter:
  `$ jtvars.bat`
- If you are using a bash or sh shell, run the `jtvars.sh` script in the Jtest installation directory. For example,
  ```
  $ cd <jtest-home>
  $ . jtvars.sh
  ```
- If you are using a csh, tcsh, or ksh shell. source the `jtvars` script in the Jtest installation directory. For example,
  ```
  $ cd <jtest-home>
  $ source jtvars
  ```
- To determine which shell you are using, enter
  `$echo $SHELL`

Creating Test Classes

Test Classes should specify test cases by using a public static void method for each test case. The correct behavior of the class should be specified using “assert (String message, boolean condition)” statements.

For example:

```
public class TestVector extends jtest.TestClass
{
    public static void testSize ()
    {
        Vector vector = new Vector ();
        vector.addElement(“name”);
        assert ("should be 1", vector.size () == 1);
```

```
}
```

Jtest will consider methods within the class that are not static public methods to be helper methods. These methods can be called to do initialization for the test case.

A Test Class can contain any number of test cases.

For examples of Test Classes, see <jtest_install_dir>/examples/dynamic/testclasses.

Naming Conventions

It is convenient to name Test Classes so that the word “Test” appends the class name. For example, if you want to create a Test Class for the class called “foo”, you should name your Test Class “fooTest”. You should also save this Test Class in the same package as the class under test. If you follow both of these recommendations and have set your CLASSPATH as described above, Jtest will automatically locate the Test Class when you start a test.

For example, if you test the java.util.Vector class, Jtest will automatically search on the CLASSPATH for a java.util.VectorTest class. If it finds this class, it will use it as the Test Class for the java.util.Vector class.

You may also use a different name and/or save the class in a package other than the one that contains the class under test; the only requirement is that the class is on your CLASSPATH. However, if you use a different name or package, you need to manually indicate that Jtest should use this Test Class (as described below in Loading Test Classes).

Defining Stubs within a Test Class

When you are using a Test Class, you can define stubs for each test method by defining a "stubs()" methods within the Test Class. For example, to specify the stubs for a test case defined by a "testXYZ" method, define a method of the form:

```
Object stubsXYZ (Method method, ...);
```

in that Test Class.

If a Test Class does not define a stubs method, or if it does not return any stubs, Jtest will apply the Class and Project Test Parameters "stubs()" methods.

For more information on User-Defined Stubs, see “Using Custom Stubs” on page 98

Using JUnit Test Classes

If you want to use a JUnit Test Class with Jtest, you need to:

- Include the junit.jar file on your CLASSPATH.
- Manually indicate the location of your Test Class (as described in Loading Test Classes below).

After you perform these steps, Jtest will use the JUnit Test Class when you run your test in the normal manner.

JUnit 3.5 is supported.

Note: The Jtest Tutorial contains a lesson on using JUnit Test Classes with Jtest. To reach this tutorial, choose **Help> Tutorial** in either Jtest UI.

Loading Test Classes

If you name your test <classname>Test, saved it in the same package as the class under test, and set your CLASSPATH, Jtest will automatically find and load it (this is described in the above section).

To manually indicate which Test Class(es) Jtest should use for the current class:

1. Right-click the Class Test Parameter tree’s **Dynamic Analysis> Test Case Generation> User Defined> Test Classes** node. A shortcut menu will open.
2. Choose **Add Test** from the shortcut menu.
3. Enter the name of your Test Class in the text field.

Checking Test Classes

If you want to check your test class:

1. Control-click the Class Test Parameter tree's **Dynamic Analysis> Test Case Generation> User Defined> Test Classes** node.
2. Choose **Check** from the shortcut menu.

Running Test Classes

Within Jtest

When Jtest performs dynamic analysis, it will execute all Test Classes that it finds automatically and/or you load manually.

If any assertion fails, Jtest will report an error in the **Specification and Regression Errors** branch of the Results panel or Errors Found panel.

If any test case throws an uncaught runtime exception, Jtest will report an error in the **Uncaught Runtime Exceptions** of the Results panel or Errors Found panel.

Note: When Jtest executes a test case within the test class, all of the Test Class's static variables will be initialized with default values.

Outside of Jtest

You can run a Test Class outside of the Jtest environment by entering the following command at the command line:

```
java jtest.TestClass <your TestClass>
```

For example, if your Test Class was named fooTest, you would enter the following command:

```
java jtest.TestClass fooTest
```

Important: In order to execute this command, you must have the jtest.zip file in your CLASSPATH.

Testing Projects That Include Test Classes

If you are testing a project that includes Test Classes, by default, Jtest will not perform static or dynamic analysis on any class that it recognizes as a Test Class.

Testing a Test Class

By default, Jtest does not perform static or dynamic analysis on any class that it recognizes as a Test Class. If you want Jtest to test a Test Class, open the class's Class Test Parameters window, then enable the **Common Parameters> Test the Test Class itself** option.

Related Topics

"About Black-Box Testing" on page 113

"Performing Black-Box Testing" on page 115

"Adding Method Inputs" on page 119

"Specifying Imports" on page 132

"Jtest Tutorials" on page 268

Specifying Imports

To specify import statements shared by all of the code used in the test specification:

1. In the Class Test Parameters window, open **Dynamic Analysis> Test Case Generation> Common**.
2. Double-click the **Imports** node. The Imports window will open.
3. Enter the import statement in the Imports window.

   Example:
   ```
   import java.util.Vector;

   import java.awt*;
   ```
4. (Optional) If you want to check the method, choose **Options> Save & Check**.
5. Choose **Options> Save**.
6. Choose **Options> Quit**.

Related Topics

“About Black-Box Testing” on page 113

“Performing Black-Box Testing” on page 115

“Adding Method Inputs” on page 119

“Adding Test Classes” on page 125

“Specifying Imports” on page 132

Using Design by Contract With Jtest

Benefits of Using DbC With Jtest

You do not need to use Design by Contract (DbC) in order to use Jtest. You can, however, increase Jtest's functionality if you use DbC; there are several main advantages to using DbC with Jtest:

- Jtest will automatically create black-box test cases that verify the functionality described in your DbC contracts (i.e., Jtest will find inputs that violate the preconditions, postconditions, class invariant contracts, and assert clauses included in the contract).
- Jtest will automatically suppress errors for inputs that violate the preconditions of the methods under test.
- Jtest will automatically suppress expected uncaught runtime exceptions that are documented using the @exception Javadoc tag.

Jtest and Jcontract

Jtest contains all necessary elements to understand DbC comments and create test cases that check whether the specifications detailed in those comments are indeed implemented. It uses the DbC information to check that the unit in and of itself is implemented correctly.

Jcontract is a new Java development tool that checks DbC contracts at runtime; it is run independently of Jtest, but the two tools are complementary. After you have used Jtest to thoroughly test a class or component at the unit level, instrument it with Jcontract, integrate it into your system/application, then Jcontract will automatically check whether its contracts are violated at runtime. Jcontract is particularly useful for determining whether an application misuses specific classes or components.

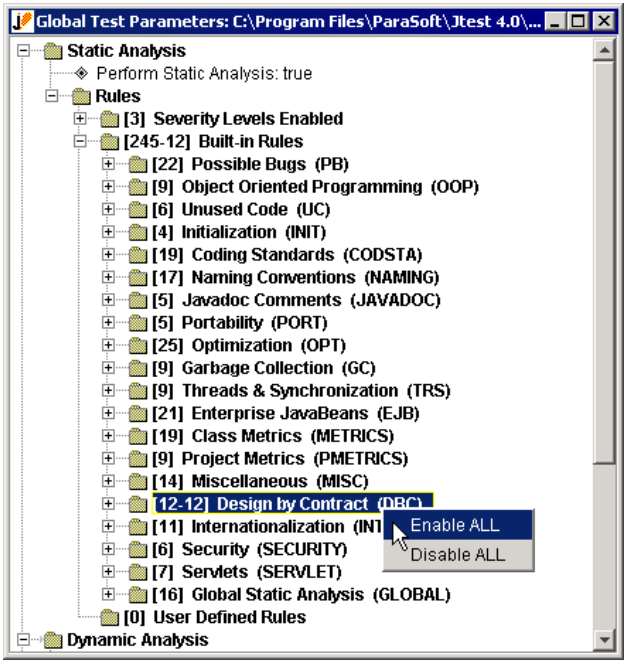
Creating DbC Comments

See for a general description of DbC, see “About Design by Contract” on page 137.

See “The Design by Contract Specification Language” on page 141 for information on how to add DbC comments to your code.

Jtest’s Design by Contract Static Analysis rules help you create well-formed DbC contracts. These rules are not enabled by default. To enable them, perform the following steps:

1. Open the Global Test Parameters window by clicking **Global**.
2. Right-click the **Static Analysis> Rules> Built-in Rules> Design by Contract** node. A shortcut menu will open.
3. Choose **Enable All** from the shortcut menu.



Using DbC Information in Tests

Jtest will use DbC information in its tests as long as the class under test's DbC contracts have been instrumented (by default, they are instrumented as the class is loaded). Just run the test in the normal manner, then Jtest reads specification information built into the class with the DbC language, and automatically develops test cases based on this specification. Jtest designs its black-box test cases as follows:

- If the code has postconditions, Jtest creates test cases that verify whether the code satisfies those conditions.
- If the code has assertions, Jtest creates test cases that try to make the assertions fail.
- If the code has invariant conditions (conditions that apply to all of a class's methods), Jtest creates test cases that try to make the invariant conditions fail.
- If the code has preconditions, Jtest tries to find inputs that force all of the paths in the preconditions.
- If the method under test calls other methods that have specified preconditions, Jtest determines if the method under test can pass non-permissible values to the other methods.

If Jtest finds inputs that violate preconditions, postconditions, class invariant conditions, and assert clauses, it will report them in the **Design by Contract Violations** branch of the Errors Found Panel (if you tested a single class) or the Results Panel (if you tested a project).

Jtest also uses the DbC information to automatically suppress exceptions that are not relevant to the project at hand. If you document valid exceptions in the code using the @exception comment tag, Jtest will suppress any occurrence of that particular exception. If you use the @pre comment tag to document the permissible range for valid method inputs, Jtest will suppress errors found for inputs that fall outside of that range.

Example Files

Example DbC files are contained in the <jtest_installation_dir>/examples/dynamic/dbc directory.

Instrumentation Options

If you use DbC comments, Jtest will by default instrument the comments when it loads the class or project under test. You can control whether or not Jtest instruments DbC comments with the **Automatically Instrument Design by Contract Comments** option in all test parameters' **Dynamic Analysis> Test Case Execution** branch.

Related Topics

"About Design by Contract" on page 137

"The Design by Contract Specification Language" on page 141

"Performing Black-Box Testing" on page 115

"Customizing White-Box Testing" on page 112

"Testing A Class - Two Simple Examples" on page 23

"Using Design by Contract With Jtest" on page 133

About Design by Contract

Design by Contract is a structured way of writing comments to define what code should do. The contract requires components of the code (such as classes or methods) to follow certain specifications as they interact with each other. The interactions between these components must fulfill a set of predetermined mutual obligations.

Design by Contract originated in Eiffel. Eiffel classes are components that cooperate through the use of the contract, which defines the obligations and benefits for each class. DbC is not yet commonly a part of programming languages such as C, C++, and Java, but ideally it should be. After all, any piece of code in any language has implicit contracts attached to it. The simplest example of an implicit contract is a method to which you are not supposed to pass `null`. If this contract is not met, a `NullPointerException` will occur. Another example is a component whose specification states that it only returns positive values. If it occasionally returns negative values and the consumer of this component is expecting the functionality described in the specification (only positive values returned), this contract violation could lead to a critical problem in the application.

Tools like Jtest and Jcontract bring Design by Contract to Java by helping you specify the contracts in comments and check whether or not the contract has been fulfilled.

Example

This is an example of a class with Design by Contract comments.

```
public class ShoppingCart
{
    /**
     * @pre item != null
     * @post $result > 0
     */

    public float add (Item item) {
        _items.addElement (item);
        _totalCost += item.getPrice ();
```

```
        return _totalCost;
    }
    private float _totalCost = 0;
    private Vector _items = new Vector ();
}
```

The contract specifies:

1. A precondition ("@pre item != null") which specifies that the item to be added to the shopping cart shouldn't be "null".
2. A postcondition ("@post $result > 0") which specifies that the value returned by the method should always be greater than 0.

Preconditions and postconditions can be thought of as sophisticated assertions. Preconditions are conditions that the client of the method needs to satisfy in order for the method to work properly. Postconditions are conditions that the implementor of the class guarantees will always be satisfied.

Benefits

Benefits of using DbC include:

- The code's assumptions are clearly documented (for example, you assume that `item` should not be `null`). Design concepts are placed directly in the code itself.
- The code's contracts can be checked for consistency because they are explicit.
- The code is much easier to reuse.
- The specification will never be lost.
- When you see the specification while writing the code, you are more likely to implement the specification correctly.
- When you see the specification while modifying code, you are much less likely to introduce errors.

Once you start using Jtest and Jcontract, the benefits of using DbC also include:

- Black-box test cases are created automatically. If you currently create your black-box test cases manually, this means fewer resources spent creating test cases and more resources you can dedicate to more complex tasks, such as design and coding. If you do not currently perform black-box testing, this will translate to more reliable software/components.
- Black-box test cases are automatically updated as the code's specification changes.
- Class/component misuse is automatically detected.
- The class implementation can assume that input arguments satisfy the preconditions, so the implementation can be simpler and more efficient.
- The class client is guaranteed that the results will satisfy the post-conditions.

For More Information

For more information about DbC see:

- Interactive Software Engineering, "Building Bug-Free O-O Software: An Introduction to Design by ContractTM." http://www.eiffel.com/doc/manuals/technology/contract/page.html
- Eldridge, G. "Java and `Design by Contract.'" http://www.elj.com/eiffel/feature/dbc/java/ge/
- Kolawa, A., "Automating the Development Process." *Software Development,* July 2000. http://www.sdmagazine.com
- Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 2000.

Note: "Design by Contract" is a trademark of Interactive Software Engineering.

Related Topics

"Using Design by Contract With Jtest" on page 133

"The Design by Contract Specification Language" on page 141

“Performing Black-Box Testing” on page 115
“Customizing White-Box Testing” on page 112
“Testing A Class - Two Simple Examples” on page 23

The Design by Contract Specification Language

This document describes the syntax and semantics for the Design by Contract (DbC) specification supported by Jtest and Jcontract.

The Design by Contract contracts are expressed with Java code embedded in Javadoc comments in the .java source file.

This document is divided into the following sections:

Tags Used for Design by Contract

The reserved Javadoc tags for DbC are:

- @invariant: Specifies class invariant condition.
- @pre: Specifies method precondition.
- @post: Specifies method postcondition.
- @concurrency: Specifies the method concurrency.

Other tags supported by Jtest and Jcontract include:

- @throws/@exception: Used to document exceptions.
- @assert: Used to add assertions in the method bodies.
- @verbose: Used to add verbose statements to the method bodies. (Not currently used by Jtest)

The following subsections describe each DbC tag in detail.

@pre

Description

Pre-conditions check that the client calls the method correctly.

Point of execution

Right before calling the method.

Scope

Can access anything accessible from the method scope except local variables. For example, it can access method arguments, and methods/fields of the class.

@post

Description

Post-conditions check whether the method works correctly.

Sometimes when a post-condition fails it means that the method was not actually supposed to accept the arguments that were passed to it. The fix in this case is to strengthen the precondition.

Point of execution

Right after the method returns successfully. Note that if the method throws an exception the @post contract is not executed.

Scope

Same as @pre, plus it can access "$result" and "$pre (type, expression)".

Accessibility

Same as @pre.

@invariant

Description

Class invariants are contracts that the objects of the class should always satisfy.

Point of execution

Same as @pre/@post: invariant checked before checking the precondition and after checking the postcondition.

Done for every non-static, non-private method entry and exit and for every non-private constructor exit.

If a constructor throws an exception, its @invariant contract is not executed.

Not done for "finalize ()".

When inner class methods are executed, the invariants of the outer classes are not checked.

Scope

Class scope, can access anything a method in the class can access, except local variables.

Accessibility

Same as @pre/@post.

@concurrency

Description

The @concurrency tag specifies how the method can be called by multiple threads. Its possible values are:

- **Concurrent:** The method can be called simultaneously by different threads (i.e., the method is multi-thread safe). Note that this is the default mode for Java methods.
- **Guarded:** The method can be called simultaneously by different threads, but only one will execute it in turn, while the other threads will wait for the executing one to finish. In other words, it specifies that the method is synchronized. Jcontract will only

report a compile-time error if a method is declared as “guarded” but is not declared as “synchronized”.

- **Sequential:** The method can only by executed by one thread at once and it is not declared synchronized. It is thus the responsibility of the callers to ensure that no simultaneous calls to that method occur. For methods with this concurrency contract, Jcontract will generate code to check if they are being executed by more than one thread at once. An error will be reported at runtime if the contract is violated.

Point of execution

Right before calling the method.

@throws/@exception

These are the standard @throws and @exception tags found in Javadoc; they are used to document that the method throws a given exception. @throws and @exception are synonymous. In this entry, we use @throws to represent both tags.

The syntax for the @throws tag is:

```
ThrowsContract
    : @throws ExceptionName Text
```

Example:

```
/** @throws NegativeArraySizeException if size is negative */
```

When a method throws an exception, the Jcontract Runtime Handler will call 'documentedExceptionThrown (Throwable t)' if that exception is documented with a @throws tag.

Note that the Runtime Monitors provided with Jcontract don't take any action when 'documentedExceptionThrown' is called. You can nevertheless take a specific action by defining a user defined Runtime Handler.

Jtest suppresses exceptions that are documented with the @throws tag as long as the the classes were instrumented with the instrument @throws condition preference set to “true”.

@assert

Syntax

The syntax for the @assert tag is:

```
AssertStmt
        : @assert BooleanExpression
        | @assert '(' BooleanExpression ')'
        | @assert '(' BooleanExpression , MessageExpression ')'
```

The MessageExpression can be of any type.

For example:

```
/** @assert value > 0 */
/** @assert (value > 0) */
/** @assert (value > 0, "value should be positive */
/** @assert (value > 0, value) */
```

The @assert tags should appear in Javadoc comments inside the method bodies. If the classes are compiled with 'dbc_javac' and the Instrument.InstrumentAssertConditions preference is true/enabled, then the @assert boolean expression will be evaluated. If the expression evaluates to false, then one or more of the following actions take place:

- An error message is reported in Jtest's **Design by Contract> @assert** Results panel/Errors Found panel branch or in the Jcontract Monitor.
- A runtime exception (jcontract.AssertException) is thrown.
- The program exits by invoking System.exit (1).

See “Contract Semantics” on page 148 for more information about how to select the actions that take place. The default action is to report an error and continue program execution.

@verbose

The syntax for the @verbose tag is:

```
VerboseStmt
        : @verbose MessageExpression
```

```
        | @verbose '(' MessageExpression ')'
```

For example:

```
    /** @verbose "process starts" */
    /** @verbose ("process ends") */
    /** @verbose 26.7 */
```

The @verbose tags should appear in Javadoc comments inside the method bodies. If the classes are compiled with 'dbc_javac' and the Instrument.InstrumentVerboseConditions preferences is true/enabled, then the classes are instrumented with the verbose expression.

By default, all verbose statements are inactive; once they are activated, they print the MessageExpression to System.out.

The @verbose statements can be separately activated for each class.The @verbose statements for a class are active if the system property jcontract.verbose.CLASSNAME is set to the value ON (where CLASSNAME is the name of the class without the package part). For example, to activate the verbose statements in class pkg.DataDictionary on Windows use:

```
$ java -Djcontract.verbose.DataDictionary=ON ...
```

Note that the MessageExpression in a verbose statement is not evaluated if the verbose statement is inactive.

Contract Syntax

The general syntax for a contract is:

```
        DbcContract:
              DbcTag DbcCode
            | @concurrency { concurrent | guarded | sequential }
```

where

```
        DbcTag:
              @invariant
            | @pre
            | @post
```

```
DbcCode:
      BooleanExpression
    | '(' BooleanExpression ')'
    | '(' BooleanExpression ',' MessageExpression ')'
    | CodeBlock
    | $none

MessageExpression:
      Expression
```

Any Java code can be used in the DbcCode with the following restriction: the code should not have side effects (i.e., it should not have assignments or invocation of methods with side-effects).

The following extensions to Java (DbC keywords) are allowed in the contract code:

- $result: Used in a @post contract, evaluates to the return value of the method.
- $pre: Used in a @post contract to refer to the value of an expression at @pre-time. The syntax to use it is:
  `$pre (ExpressionType, Expression).`
  Note: The full "$pre (...)" expression should not extend over multiple lines.
- $assert: Can be used in DbcCode CodeBlocks to specify the contract conditions.
  The syntax to use it is:
  `$assert (BooleanExpression)`
  or
  `$assert (BooleanExpression , MessageExpression)`
- $none: Used to specify there is no contract.

Notes

- The @pre, @post and @concurrent tags apply to the method that follows in the source file.
- The MessageExpression is optional and will be used to identify the contract in the error messages or contract violation exceptions thrown. The MessageExpression can be of any type. If it is a

reference type it will be converted to a String using the "toString ()" method. If it is of primitive type it will first be wrapped into an object.

- There can be multiple conditions of the same kind for a given method. If there are multiple conditions, all conditions are checked. The conditions are ANDed together into one virtual condition. For example it is equivalent (and encouraged for clarity) to have multiple @pre conditions instead of a single big @pre condition.

Examples

```
    /**
      * @pre {
      *     for (int i = 0; i < array.length; i++)
      *         $assert (array [i] != null, "array elements
                   are non-null");
      * }
      */

    public void set (int[] array) {...}

    /** @post $result == ($pre (int, arg) + 1) */

    public int inc (arg) {...}

/** @invariant size () >= 0 */

    class Stack {...}

    /**
     * @concurrency sequential
     * @pre (value > 0, "value positive:" + value)
     */

    void update (int value) {...}
```

Contract Semantics

The contracts are specified in comments and will not have any effect if compiling or executing in a non DbC enhanced environment.

In a DbC-enhanced environment, the contracts are executed/checked when methods of a class with DbC contracts are invoked.

A contract fails if any of these conditions occur:

- The "BooleanExpression" evaluates to "false."
- An "$assert (BooleanExpression)" is called in a "CodeBlock" with an argument that evaluates to "false."
- The method is called in a way that violates its @concurrency contract.

If a contract fails, the Runtime Handler for the class is notified of the contract violation. Jcontract provides several Runtime Handlers; the default one uses a GUI Monitor that shows program progress and contract violations. You can also write your own Runtime Handlers.

With the Monitor Runtime Handlers provided by Jcontract, program execution continues as if nothing has happened when a contract is violated. For example, if a @pre contract is violated, the method will still be executed.

This option makes the DbC-enabled and non DbC-enabled versions of the program work in exactly the same way. The only difference is that in the DbC-enabled version, the contract violations are reported to the current Jcontract Monitor.

Note: Contract evaluation is not nested; when a contract calls another method, the contracts in the other method are not executed.

Contract Inheritance

Contracts are inherited. If the derived class or overriding method doesn't define a contract, it inherits that of the super class or interface. Note that a contract of $none implies that the super contract is applied.

If an overriding method does define a contract then it can only:

- Weaken the precondition: Because it should at least accept the same input as the parent, but it can also accept more.
- Strengthen the postcondition: Because it should at least do as much as the parent one, but it can also do more.

To enforce this:

- When checking the @pre condition, the precondition contract is assumed to succeed if any of the @pre conditions of the chain of overridden methods succeeds (i.e., the preconditions are ORed).
- When checking the @post condition, the postcondition contract is assumed to succeed if all the @post conditions of the chain of overridden methods succeed (i.e., the postconditions are ANDed).

Note: If there are multiple @pre conditions for a given method, the preconditions are ANDed together into one virtual @pre condition and then ORed with the virtual @pre conditions for the other methods in the chain of overridden methods.

For @invariant conditions, the same logic as for @post applies.

@concurrency contracts are also inherited. If the overriding method doesn't have an @concurrency contract, it inherits that of the parent. If it has an inheritance contract, it can only weaken it (as it does for @pre conditions). For example, if the parent has a “sequential” @concurrency, the overriding method can have a “guarded” or “concurrent” @concurrency.

Coding Conventions

When using Design by Contract in Java, the following coding conventions are recommended:

- Place all the @invariant conditions in the class Javadoc comment with the Javadoc comment appearing immediately before the class definition.
- Javadoc comments with the @invariant tag should appear before the class definition.
- All public and protected methods should have a contract. All package-private and private methods should also have a contract.
- If a method has a DbC tag, it should have a complete contract. This means that if you have both a precondition and a postcondi-

tion, you should use "DbcTag $none" to specify that a method doesn't have any condition for that tag.

- No public class field should participate in an @invariant clause. Because any client can modify such a field arbitrarily, there is no way for the class to ensure any invariant on it.
- The code contracts should only access members visible from the interface. For example, the code in a method's @pre condition should only access members that are accessible from any client that could use the method. In other words, the contract of a public method should only use public members from the method's class.

Note: Jcontract does not currently enforce these conventions.

Related Topics

"Using Design by Contract With Jtest" on page 133

"About Design by Contract" on page 137

"Performing Black-Box Testing" on page 115

"Customizing White-Box Testing" on page 112

"Testing A Class - Two Simple Examples" on page 23

About Regression Testing

Regression testing checks that class behavior doesn't change. Regression testing gives you the peace of mind of knowing that the class doesn't break when the code is modified.

Jtest provides automatic regression testing. Even if you don't specify what the correct outcomes are, Jtest remembers the outcomes from previous test runs, compares outcomes every time the class is tested, and reports an error for any outcome that changes.

Related Topics

“Performing Regression Testing” on page 153

Performing Regression Testing

Jtest performs regression testing, along with all other appropriate types of testing, each time that you test a class or set of classes that has already been tested at least once.

To perform regression testing:

1. Open the appropriate UI for your test. The Class Testing UI is used to test a single class; the Project Testing UI is used to test a set of files.
    - 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. Restore previously saved test parameters by doing one of the following:
    - Choosing **File> Open**, then selecting the appropriate .ctp (for class test parameters) or .ptp (for project test parameters) file in the file chooser.
    - (For recently accessed tests) Choosing **File> Open Recent> [File Name]**.
3. Run the test by clicking the **Start** button.

Regression errors found will be reported in the **Specification and Regression Errors** branch of the Errors Found Panel (if you tested a single class) or the Results Panel (if you tested a project).

Related Topics

"About Regression Testing" on page 152

"Jtest Tutorials" on page 268

Integrating VisualAge and Jtest

Jtest integrates into IBM's VisualAge 3.5, 3.5.3, and 4.0.

After you integrate Jtest into VisualAge, you will be able to test your files with Jtest from within the VisualAge IDE.

To integrate Jtest into VisualAge:

1. If you have not already done so, install both Jtest and VisualAge.
2. If you have not already done so, start Jtest and close VisualAge.
3. Choose **Tools> IDE Integration> IBM VisualAge> <visualage version_number>**in either of Jtest's UIs.

Jtest will then open a dialog box that displays the VisualAge installation directory it detected. If this is not the correct directory, change the directory by entering the correct directory or by browsing to it.

Next, Jtest will place the correct files in the VisualAge IDE's **Tools** directory, and you will be ready to use Jtest within VisualAge.

For details on using Jtest within VisualAge, see "Using Jtest Within VisualAge" on page 155.

Using Jtest Within VisualAge

After you have integrated Jtest into VisualAge as described in “Integrating VisualAge and Jtest” on page 154, you can use Jtest within the VisualAge IDE to perform any of the following actions:

- Test a single class within a project.
- Test a package.
- Test a project.
- Test a class that uses a class in another project.
- Edit your source code.
- Import Test Classes or Stub Classes.
- Remove exported files.

Testing a Single Class Within a Project

1. In the VisualAge IDE, right-click the class that you want to test, then choose **Tools> Jtest> Test Class** from the shortcut menu. This will export the Project that contains the class and load the class in Jtest’s Class Testing UI. The VAJ log window will indicate that it is exporting into Jtest. After the export is complete, the log window will indicate that you must export the class under test’s dependencies before you start testing in Jtest.
2. To start testing, click **Start** in Jtest’s Class Testing UI.

Testing a Package

1. In the VisualAge IDE, right-click the package that you want to test, then choose **Tools> Jtest> Test Package** from the shortcut menu. This will export the project the class is contained within and load the package path in Jtest’s Project Testing UI. The VAJ

log window will indicate that it is exporting into Jtest. After the export is complete, the log window will indicate that you must export the package under test's dependencies before you start testing in Jtest.

2. To start testing, click **Start** in Jtest's Project Testing UI.

Testing a Project

1. In the VisualAge IDE, right-click the project that you want to test, then choose **Tools> Jtest> Test Project** from the shortcut menu. This will export the project and load the project path in Jtest's Project Testing UI. After the export is complete, the log window will indicate that you must export the project under test's dependencies before you start testing in Jtest.
2. To start testing, click **Start** in Jtest's Project Testing UI.

Testing a Class that Uses a Class in Another Project

To test a class that uses a class in another project, you need to export the dependencies into Jtest. You can export a single class, a package, or a project.

To export a single class needed for testing:

- Right-click the class in the VisualAge IDE, then choose **Tools> Jtest> Export Class** from the shortcut menu.

To export a package needed for testing:

- Right-click the package in the VisualAge IDE, then choose **Tools> Jtest> Export Package** from the shortcut menu.

To export a project needed for testing:

- Right-click the project in the VisualAge IDE, then choose **Tools> Jtest> Export Project** from the shortcut menu.

When you export a class, package, or project into Jtest, it will be automatically be added to the Jtest classpath. All classes, packages, and projects

that you have tested or exported into Jtest will be placed in the following path:
<jtest_installation_directory>/u/<your_user_name>/
tmp/va/export/<path to class >

Editing Your Source Code

Whenever you choose to edit your source code from Jtest, Jtest will automatically open the appropriate file in the VisualAge IDE.

When you start a test in Jtest, the package or project being tested is re-exported to capture any changes made when you edited your code within VisualAge.

Importing Test Classes and Stub Classes

To use Test Classes and Stub Classes, you must first import them into your workspace. To do this, choose **Workspace> Tools> Jtest> Import Jtest classes** from the VisualAge menu bar. This will create a project titled “Jtest classes” and import Jtest classes (Jtest API) in VisualAge’s workspace.

Removing Exported Files

After you are done testing, if you want to remove the files that Jtest exported to your file system, choose **Workspace> Tools> Jtest> Clean Export Directory** from the VisualAge menu bar

Additional Notes on Jtest/VisualAge Integration

- Only files currently in the workspace can be exported.
- When Jtest is started, the project that is being tested (or the project that contains the class or package being tested) is automatically exported to the File System. All the classes must be in the workspace. Classes that are in the repository but not in the workspace will not be exported.
- If a class that you want to test depends on a class or package in another project, you need to perform one of the following steps to export that class, package, or project into Jtest:

    - Right-click the class, then choose **Tools> Jtest> Export Class**.
    - Right-click the package, then choose **Tools> Jtest> Export Package**.
    - Right-click the project, then choose **Tools> Jtest> Export Project**.
- If you change a tested class, package, or project while Jtest is open, you don't need to restart Jtest. When you click Jtest's **Start** button, the entire project is re-exported and all of the changes in that current project will be captured.
- Jtest behaves as follows to improve testing speed:
    - When you right-click a package or project then choose **Tools> Jtest> Test Project/Package**, Jtest asks you whether you would like to re-export the project/package. If you modified your code since you chose **Test Project/Package** (and want to test the modified code, choose **Yes**. Otherwise, choose **No**.
    - Jtest will not collect global static analysis information before you test in the Class Testing UI. This information will still be collected when you test a set of files in the Project Testing UI.

Integrating JBuilder and Jtest

Jtest integrates into Inprise's JBuilder 4 and 5.

After you integrate Jtest into JBuilder, you will be able to test your files with Jtest from within the JBuilder IDE.

To integrate Jtest into JBuilder:

1. If you have not already done so, install both Jtest and JBuilder.
2. If you have not already done so, start Jtest and close JBuilder.
3. Choose **Tools> IDE Integration> Inprise JBuilder> <jbuilder_version_number>** in either of Jtest's UIs.

Jtest will then open a dialog box that displays the JBuilder installation directory it detected. If this is not the correct directory, change the directory by entering the correct directory or by browsing to it.

Next, Jtest will place the correct files in the JBuilder IDE's **Tools** directory, and you will be ready to use Jtest within JBuilder.

For details on using Jtest within JBuilder, see "Using Jtest Within JBuilder" on page 161.

Using Jtest Within JBuilder

After you have integrated Jtest into JBuilder as described in “Integrating JBuilder and Jtest” on page 160, you can use Jtest within the JBuilder IDE to test a single class or a project.

Note: When you use Jtest from within JBuilder, results are saved in <jtest_install_dir>/u/<username>/JB.

Testing a Single Class Within a Project

1. In the JBuilder IDE, ensure that the class that you want to test is active.
2. Choose **Tools> Jtest> Test Active Class**. This will load the active JBuilder class in Jtest’s Class Testing UI.
3. To start testing, click **Start** in Jtest’s Class Testing UI.

Testing a Project

1. In the JBuilder IDE, ensure that the class that you want to test is active.
2. Choose **Tools> Jtest> Test Active Project**. This will load the active JBuilder project in Jtest’s Project Testing UI.
3. To start testing, click **Start** in Jtest’s Project Testing UI.

Editing Source Code in JBuilder

If you have integrated JBuilder into Jtest and you choose to edit your source code from Jtest, Jtest will open up the source code in JBuilder. If you choose to edit the source code related to a specific error message (for example, by right-clicking a static analysis violation message with a specific line reference, then choosing **Edit Source**), the referenced line of source code will be highlighted in JBuilder.

Saving and Restoring Tests Parameters

Saving Parameters

If you save test parameters, you will be able to instantly restore the precise testing circumstances that you used for a previous test. This lets you accurately repeat a test to see if modifications caused class behavior to change and/or introduced errors (i.e., it lets you perform regression testing).

Whenever Jtest tests a class, it automatically saves the class test parameters in the file whose name is shown in the status bar. Whenever Jtest tests a set of classes, it automatically saves project test parameters for the project as a whole, as well as class test parameters for each class contained in the project.

You can also save parameters under a different name by choosing **Save As** from the **File** menu in the appropriate UI. (Use the Project Testing UI to save project test parameters, and the Class Testing UI to save class test parameters).

Restoring Test Parameters

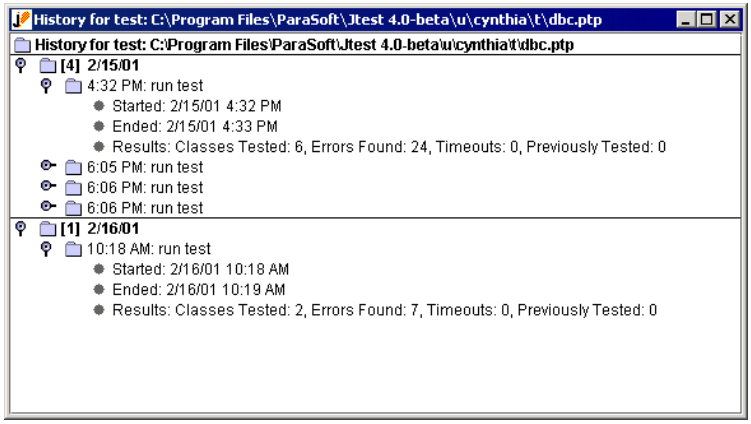
To restore parameters from a previous test (for example, to perform regression testing), simply choose **File> Open** from the appropriate UI (Use the Class Testing UI to restore class test parameters; use the Project Testing UI to restore project test parameters). In the file chooser that opens, select the .ctp (for class test parameters) or .ptp (for project test parameters) file that you want to restore, then Jtest will open the specified test parameters.

You can also open recent parameters by choosing **File> Open Recent> [File Name]**.

Viewing Test History

To view a record of all previous Project Test runs (including test start and end time, as well as a brief summary of test results) for the current project test, click the **History** button in the Project Testing UI tool bar.

To view the same type of data for every previous Project Test run, right-click the **History** button in the Project Testing UI tool bar, and choose **Global History** from the shortcut menu.



Both history windows have identical functionality.

- To remove a record from the test history, right-click the appropriate record, then choose **Delete** from the shortcut menu.
- To delete all history entries, right-click the **History** node and choose **Delete All Entries** from the shortcut menu.
- To view a log of a test run, right-click the node that identifies that test run, then choose **View Log** from the shortcut menu.
- To view a summary report of a test run, right-click the node that identifies that test run, then choose **View Summary Report** from the shortcut menu.

- To view metrics for a test run, right-click the node that identifies that test run, then choose **View Project Metrics** from the shortcut menu. For information on creating graphs that track how metrics vary as the project progresses, see “Tracking Metrics Over Time” on page 76.

Viewing Coverage Information

You can view coverage information in three areas:

- The Class Testing UI's Test Progress panel (see "Test Progress Panel" on page 201).
- The Project Testing UI Results panel's **[Class Name]> Test Progress> Dynamic Analysis> Total Coverage** branch. (see "Test Progress" on page 47).
- A single class report (see "Single Class Report" on page 177).

A method is designated "covered" if Jtest automatically tests any part of the constructor.

Jtest performs data coverage for the generated input categories; this means that the parts of the class that have been covered are thoroughly tested with respect to those inputs.

The coverage reported is relative to the classes that have been accessed for the paths Jtest has tried. If some part of the class is not covered, it means that Jtest has not yet found a path leading to those statements or no path leads to those statements.

In class testing mode, Jtest usually covers approximately 50% of a class's code. Sometimes Jtest will be able to test 100% of the class, and sometimes it will test less than 50% of the class.

Generating Coverage Information For Every Class the Original Class Accesses

To have Jtest generate coverage information for every class that the original class accesses, choose **Preferences> Configuration Options> Report file> Show All Classes Accessed**. The next report opened will

include coverage information for all classes accessed by the original class under test.

Determining What Lines Were Not Covered

To determine what lines were not covered, view the single class test report file. Any lines that have a “>” in front of them were not tested.

Viewing Context-Sensitive Help

Jtest has context-sensitive help topics associated with almost every option, command, and UI component.

To view context-sensitive help topics:

1. Enable context-sensitive help by clicking the **Context Help** button in either UI's tool bar, or by choosing **Help> Activate Context Help**.
2. Move your cursor over the item that you want to learn more about. If a context-sensitive help topic is available for this element, that topic will then open.

Viewing, Editing, or Compiling a Source

Viewing the Source of a Violation

You can view the source of a violation, with the problematic line highlighted, by double-clicking the file/line information for the error in the Errors Found panel (in the Class Testing UI) or in the Errors Found branch of the lower Results panel (in the Project Testing UI). Another way to view the source is to right-click that same error message and choose **View Source** from the shortcut menu.

Editing the Source of a Violation

You can also edit the source of a violation from within Jtest. To edit the source of a violation, right-click the file/line information for the violation message in the Errors Found panel (in Class Testing UI) or in the Errors Found branch of the lower Results panel (in Project Testing UI), and choose **Edit Source** from the shortcut menu. The source will then open in Notepad (Windows) or vi (UNIX). (To use a different editor, choose **Preferences> Configuration Options> Editor**, and enter your preferred editor in the dialog box that opens).

Viewing, Editing, and Compiling Any Source

In addition, you can view, edit, and compile the source of *any* class under test (whether or not it contains a violation) via the Class Testing UI. Before you perform any of these actions, you must first open the class in the Class Testing UI. When your class is open in the Class Testing UI, you can perform any of the following actions:

- **View class source in Jtest's Source Viewer:** Click **Source**.

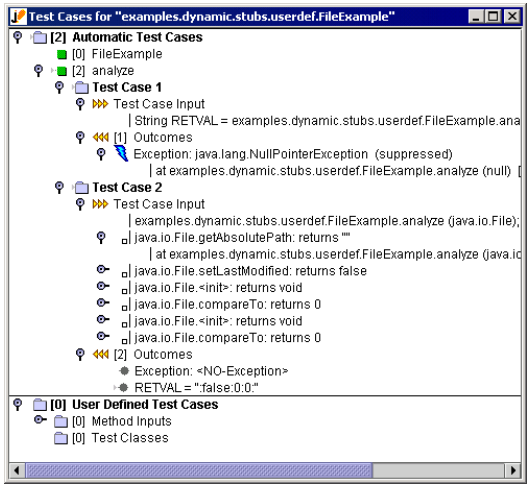
- **Edit class source in source editor:** Right-click the **Source** button, then choose **Edit Class Source**. Notepad (Windows) or vi (UNIX) is used as the default editor. To use a different editor, choose **Preferences> Configuration Options> Editor**, then enter your preferred editor in the dialog box that opens.
- **Compile the current class:** Right-click the **Source** button, then choose **Compile Class**.

Viewing and Validating Test Cases

In the View Test Cases window, you can view and validate the test cases that Jtest used for Dynamic Analysis.

To open this window from the Class Testing UI, click the **View Test Cases** button.

To open this window from the Project Testing UI's Results panel, right-click the **[Class Name]** node, then choose **View Test Cases** from the shortcut menu.



About the View Test Cases Window

The View Test Cases window contains the following nodes:

Test cases for [classname]

Contains the test cases that Jtest generated and executed in this

class's most recent test.

Automatic Test Cases

Contains the test cases that Jtest generated automatically. Only the test cases that do something new (e.g., increase coverage, throw a new exception, etc.) are shown.

[method name]

Contains test cases for this method.

Test Case

Contains all of the information for a test case.

Test Case Input

Contains input that defines the test case.

The input for automatic test cases is the calling sequence.

The input for user defined test cases is the input for each argument.

If stubs were used, they will be listed here. Empty boxes indicate automatically generated stubs. Black boxes indicate user-defined stubs. For more information on stubs, see "Testing Classes That Reference External Resources" on page 93 and "Using Custom Stubs" on page 98.

Outcomes

Contains outcomes for this test case. Verify if the outcomes are correct or incorrect according to the class specification and set their state using the shortcut menus.

When the outcome is an object, Jtest automatically chooses the toString method to show its state.

If a method named jtestInspector is defined for the object's class, Jtest will only use the return value of this method to show the object state.

If no toString or jtestInspector methods are defined, Jtest will heuristically choose some public instance methods for that object to show its state.

If the method under test is a static method, Jtest will heuristically choose public static methods to show the class state. If the methods Jtest chose are not enough, declare a static method called sjtestInspector for the class. Jtest will use the return value of this method to show the object class.

[n]= number of outcomes for this test case.

Exception

Indicates whether an exception occurred, and, if so, what type of exception occurred.

If an exception was suppressed, you can see the reason for the suppression by right-clicking the exception message node and choosing **Why Suppressed?** from the shortcut menu.

User Defined Test Cases

Contains test cases generated from user-defined input.

Method Inputs

Contains test cases generated from method inputs.

[method name]

Contains test cases for this method.

Test Case

Contains all of the information for a test case.

Test Case Input

Contains input that defines the test case.

The input for automatic test cases is the calling sequence.

The input for user defined test cases is the input for each argument.

If stubs were used, they will be listed here. Empty boxes indicate automatically generated stubs. Black boxes indicate user-defined stubs. To see the stack trace where a stub invocation occurred, expand the stub's branch. For more information on stubs, see "Testing Classes That Reference External Resources" on page 93 and "Using Custom Stubs" on page 98.

Outcomes

Contains outcomes for this test case. Verify if the outcomes are correct or incorrect according to the class specification and set their state using the shortcut menus.

When the outcome is an object, Jtest automatically chooses the toString method to show its state.

If a method named jtestInspector is defined for the object's class, Jtest will only use the return value of this method to show the object state.

If no toString or jtestInspector methods are defined, Jtest will heuristically choose some public instance methods for that object to show its state.

If the method under test is a static method, Jtest will heuristically choose public static methods to show the class state. If the methods Jtest chose are not enough, declare a static method called sjtestInspector for the class. Jtest will use the return value of this method to show the object class.

[n]= number of outcomes for this test case.

Exception

Indicates whether an exception occurred, and, if so, what type of exception occurred. When an exception occurs, stack trace information can be displayed by opening this node.

If an exception was suppressed, you can see the reason for the suppression by right-clicking the exception message node and choosing **Why Suppressed?** from the shortcut menu.

Test Classes

Contains the number of test cases added from test classes.

If you change specification and regression test cases and want to restore the set used during the actual tests, right-click the **Specification and Regression Test Cases** node, then choose the **Reload** option from the shortcut menu. Jtest will then reload the original test cases.

The color of the arrow to the left of a leaf has the following meaning:

- **green:** The outcome is correct, or has been validated as correct by the user.
- **red:** The outcome is incorrect (or has been validated as incorrect by the user), or an uncaught runtime exception was detected.
- **gray:** The outcome status is unknown, and no uncaught runtime exceptions were detected.
- **no arrow:** The user has specified to ignore this outcome.

If the **Perform Automatic Regression Testing** flag is selected, Jtest will assume that gray outcomes are correct and will report an error if the outcome changes.

In this window, the outcome is marked as incorrect if it is different than the one in the **Specification and Regression Test Cases** branch of the Errors Found panel (in the Class Testing UI) or Results panel (in the Project Testing UI). When more than one test case outcome differs, only one of them is marked as an error and only that one is reported as an error in the Errors Found panel or Results panel.

Validating Outcomes

Indicate whether or not the outcome for each test case is correct by right-clicking the appropriate outcome node, then choosing **Mark as Cor-**

rect (if the listed outcome is the expected outcome), **Mark as Incorrect** (if the listed outcome is not the expected outcome), **Mark as Unknown** (if you don't know how the listed outcome compares to the expected outcome), or **Mark as Ignore** (if you want Jtest to ignore the listed outcome) from the shortcut menu.

To ignore an entire test, right-click the appropriate **Test Case** node, and choose **Ignore this Test Case** from the shortcut menu. To tell Jtest to stop ignoring a test case you previously told it to ignore, right-click the appropriate **Test Case** node, and choose **Do Not Ignore this Test Case** from the shortcut menu.

To evaluate all of a test case's outcomes with one click, right-click the appropriate **Outcomes** leaf, then choose the appropriate **Set All to...** command from the shortcut menu.

To remove an entire test case, right-click the appropriate **Test Case** node, then choose **Delete** from the shortcut menu.

To remove the entire set of test cases, right-click the **Specification and Regression Test Cases** node, then choose **Delete All** from the shortcut menu.

To indicate the correct outcome for a test case:

1. Open the Class Test Parameters window.
2. Open that test case's branch in **Dynamic Analysis> Test Case Evaluation> Specification and Regression Testing**.
3. Right-click the outcome, choose **Edit** from the shortcut menu, then enter the correct value in the text field that opens.

Viewing a Report of Results

Jtest generates the following types of reports:

- Single Class Report
- Project Report
- Detailed Project Report
- Summary Project Report

These reports all use the standard JNI 1.1 specification to identify methods.

By default, the reports are formatted in text (ASCII) format. If you would like Jtest to generate HTML reports (e.g., if you want to post the report on your development intranet), choose **Preferences> Configuration Options> Report Format> HTML**.

Single Class Report

After performing a test on a single class, Jtest will generate a Single Class Report of the testing session.

This report contains, among other information, the annotated source code for the tested class. This may be used to determine what lines Jtest tested and what lines it did not test. Lines of code that were not tested are marked with a “>” at the beginning of the line; absence of the “>” indicates that a line was tested.

To access the Single Class Report, click the **Report** button in the Class Testing UI tool bar.

By default, Jtest does not show the source of every accessed class unless you tell it to do so. Use the **Preferences> Configuration Options> Report File> Show All Classes Accessed** option to tell Jtest to display the annotated source of each class accessed by the class under test. Also, Jtest does not, by default, show the test cases used for the test. Use the **Preferences> Configuration Options> Report File> Show Test Cases** option to tell Jtest to include test case information in this report.

If you would like a Single Class Report for a class included in a project test, open the class in the Class Testing UI, then click the **Report** button.

Summary Project Report

This is the least detailed of the three available project reports. This report contains the test name and a one line report of each error available in the Results panel; if there are no errors, this report contains only the test name.

To access the Summary Project Report, right-click the **Report** button in the Project Testing UI tool bar, then choose **View Summary Report** from the shortcut menu; you can also obtain this report by using `-summary_report` in the command line.

All project reports contain only information on the classes and errors that were available in the Results panel when the **Report** button was clicked. To limit the classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **Report** button.

Project Report

This report is more detailed than the Summary Project Report, but less detailed than the Detailed Project Report. This report contains project test parameters as well as all essential details about each error available in the Results panel.

To access the Project Report, click the **Report** button in the Project Testing UI tool bar, or use `-report` at the command line.

All project reports contain only information on the classes and errors that were available in the Results panel when the **Report** button was clicked. To limit the classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **Report** button.

Detailed Project Report

This is the most detailed of the three available project reports. This report contains project test parameters, class test parameters, and all results information available in the Results panel.

To access the Detailed Project Report, right-click the **Report** button in the Project Testing UI tool bar, then choose **View Detail Report** from the shortcut menu; you can also obtain this report by using `-detail_report` in the command line.

All project reports contain only information on the classes and errors that were available in the Results panel when the **Report** button was clicked. To limit the classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **Report** button.

Customizing Test Parameters

The testing done by Jtest is highly customizable. Test parameters can be customized at three levels:

- Global Test Parameters (apply to all tests)
- Project Test Parameters (apply to a specific project, or set of classes)
- Class Test Parameters (apply to a specific class)

For details on each of these types of parameters, see the appropriate topics.

In general, you modify parameter values by right-clicking and choosing available options from the shortcut menus that open, by double-clicking a node and entering values in a dialog box that appears, or by selecting/clearing the radio button to the left of a node.

Parameters that Appear at Multiple Levels

When setting parameters, be aware that several parameters appear at more than one level. For example the parameter **Static Analysis> Rules> Severity Levels** appears in all parameters (Global, Project and Class).

When testing a class, Jtest uses the value in the Class Test Parameters. If a value is set to **Inherit**, the value of the current parent parameter is used. In this case, the actual value that will be used is shown in the icon or in parentheses.

When creating a new test, the value in the parent parameter is used. For example, when creating a Project test, the value of the flag in the Global Test Parameters is used as the initial value. In such instances (where the child parameter inherits a value from a parent parameter) the parameter value is designated as **Inherit**.

Sharing Project Test Parameters

You can have multiple members of your team run identical tests or subsets of a test by sharing test parameters. Parameters can be shared by anyone running Jtest, whether they are on the same machine or a different machine. For example, if you create a project and add some test cases to some of your classes, any other team member who shares your project parameters can edit any of your project's classes, then run the exact same tests that you ran.

The following steps explain how to share project test parameters; they are best understood when you are looking at the example project contained in dv_mcjt.zip:

1. In the Project Testing UI, create a .ptp file (e.g., all.ptp) in source control that contains the project test parameters.
    a. In the Project Testing UI's **Search In** field, enter the location of the classes that you want to test. For example, enter

       `$HOME/dv/mcjt`.

    b. Use the Project Test Parameters' **Common Parameters> Directories> Class Test Parameters Root** node to specify a class test parameters root that falls under source control.

       For example, to tell Jtest to put each .ctp file in the same place as the associated .java file, open the Project Test Parameters window, go to **Common Parameters> Directories**, then enter

       `$HOME/dv/mcjt`

       in the **Class Test Parameters Root** node.

    c. Specify any other Project Test parameters that may be needed (e.g., parameters in **Common Parameters> java/javac-like Parameters**, or in **Common Parameters> Source Path**).

d. Use **File> Save As** to save the project test parameters in a location under source control. For example, save the project as all.ptp in the $HOME/dv/mcjt directory

2. Create specific .ptp files to test subprojects within the full project.

   a. Each developer in the group should open the .ptp file for the full project, modify the **Filter in** field's parameter so that Jtest only tests the classes he or she is responsible for, then save the modified project as a different .ptp file.

      For example, to break frog.ptp into frog_yellow.ptp and frog_green.ptp:

      - Copy frog.ptp to frog_yellow.ptp.
      - Modify the value in the **Filter In** field to animals.amphibians.frog.yellow
      - Copy frog.ptp to frog_green.ptp.
      - Modify the value in the **Filter In** field to animals.amphibians.frog.green.

   b. Each developer should work with the .ptp and .ctp files for the classes he or she is responsible for.

   Note that the first time Jtest creates each class, it automatically creates a .ctp file for it. This .ctp file will be located in the same location as the corresponding .java file. For example, if you have a class at

   `animals/mammals/human/Human.java`

   Jtest will create a .ctp file at

   `animals/mammals/human/Human.ctp`

   The .ctp files in the project that you want to share should also be placed under source control. The person responsible for a given .java file should be responsible for the corresponding .ctp file.

   You only need to modify the .ctp file when you change some of the class test parameters (e.g. if you add test cases, change outcome values, or suppress static analysis error messages). In those cases, you need to check the .ctp file in and out of source control.

3. To run all tests after the build, open the .ptp file associated with the full project test, then run the test by clicking the **Start** button. Jtest will then test all of the classes in the project using the .ctp files that all developers collaborated on.

Customizing Reporting of Violations

To customize Jtest so that it only reports the errors relevant to your project, you can suppress the reporting of any static analysis warning messages and uncaught runtime exception messages that you do not want displayed.

Related Topics

“Static Analysis Suppressions” on page 84

“Dynamic Analysis Suppressions” on page 89

“Customizing White-Box Testing” on page 112

Customizing System Settings

Use commands found in the **Preferences** menu of the Class Testing UI or Project Testing UI menu bar to customize Jtest system settings.

For more information about available menu items, see “Class Testing UI Menu Bar” on page 190 and “Project Testing UI Menu Bar” on page 206.

Jtest UI Overview

Jtest has two available UIs:

- Class Testing UI**:** Area to test a single class or view results of a single class tested as part of a project test.
- Project Testing UI: Area to test a set of classes (from a directory, zip file, or jar file). When this UI is open, it tales control of the Class Testing UI.

By default, Jtest opens the Class Testing UI the first time that you start Jtest, then opens the last UI that you were working with all subsequent times you start Jtest.

To determine which UI appears when Jtest is started, choose **Preferences> UI Preferences> Starting UI <Desired UI>**. To configure Jtest to automatically open whichever UI you were working with the last time that you closed Jtest, choose **Last UI Visible** instead of **<Desired UI>**.

Trees

The following features are common to all of Jtest's trees:

- **Shortcut menus for the nodes:** Many of Jtest's tree nodes contain shortcut menus that allow you to perform various actions related to that node. If a tree node has an associated shortcut menu, a right-click icon will appear when your cursor is placed over that node. To access a node's shortcut menu, right-click the node. To access context-sensitive help for a certain shortcut menu option, enable context-sensitive help, then position your cursor over the shortcut menu option that you want to learn more about.
- **Shortcut menus for the trees:** All of Jtest's trees have an extra shortcut menu that you can access by clicking the right mouse button while pressing the **Control** key.
  This extra shortcut menu contains the following commands:
  - **Find:** Finds strings in the tree.
  - **Print:** Prints the tree.
  - **Expand Children:** Completely expands the tree to reveal all children.
  - **Collapse Children:** Collapses all children in the tree.
  - **Check:** Checks the node contents (if the node allows that operation) and/or displays any error messages associated with the node.

Cursors

Jtest uses two special cursors to alert you to “hidden” options and/or information:

- **The**  **help cursor:**

  After context-sensitive help has been enabled, this cursor indicates that there is a context-sensitive help topic available for the item that the cursor is positioned over.

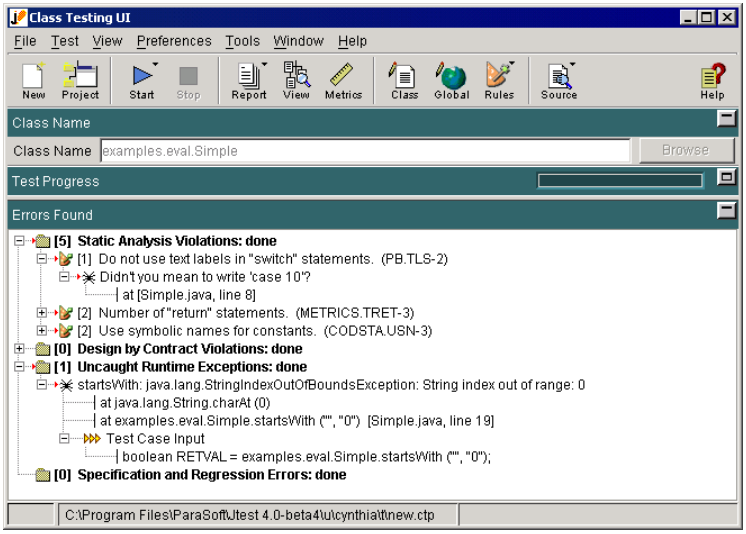
- **The**  **right-click cursor:**

  This cursor indicates that a shortcut menu is associated with the item that the cursor is positioned over. The shortcut menu can be accessed by right-clicking the item.

Class Testing UI

The Class Testing UI allows you to perform and configure tests of single classes, as well as focus on the results of a class tested as part of a project test. This UI consists of the following components:

- Class Testing UI Menu Bar
- Class Testing UI Tool Bar
- Class Name Panel
- Test Progress Panel
- Errors Found Panel



For information on testing a class in the Class Testing UI, see “Testing a Single Class” on page 21.

Class Testing UI Menu Bar

File

Commands in this menu control basic test functionality.

- **New:** Starts a new session by clearing any existing setup values or settings.
- **Open:** Opens an existing test specification saved as a .ctp file.
- **Open Recent:** Opens parameters of a recent test. Contains a list of the most recently opened classes; choose a file name from this list to open the associated parameters file. The **Clear List** command clears all items from this list.
- **Save:** Saves the current class test parameters in the test parameters file shown in the status bar.
- **Save As:** Saves the current class test parameters in the test parameters file that you specify.
- **Close UI:** Closes the Class Testing UI. If the Project Testing UI is not open, choosing this command will also close Jtest.
- **Exit:** Closes Jtest.

Test

Commands in this menu start and stop tests.

- **Start:** Starts testing the class whose name appears in the **Class Name** field.
- **Stop:** Stops the current test.

View

Commands in this menu display information related to the current test.

- **Report:** Contains the following report-related commands:

    - **View Report:** Displays the Single Class Report of the current test.
    - **Print ASCII Report:** Sends the Single Class Report directly to the printer.
- **Test Cases:** Opens the View Test Cases window (displays test cases that Jtest used for Dynamic Analysis).
- **Metrics:** Displays class metrics.
- **Class Test Parameters:** Lets you view and edit the Class Test Parameters (parameters used for the current class test).
- **Global Test Parameters:** Lets you view and edit the Global Test Parameters (parameters used for all Jtest tests).
- **Source:** Contains the following commands that let you view, find, and compile source files:
    - **View Class Source:** Displays the source of the current class in Jtest's source viewer.
    - **Edit Class Source:** Displays the source of the current class in the integrated source editor.
    - **Locate .java file:** Displays the path to the .java file currently under test.
    - **Locate .class file:** Displays the path to the .class file currently under test.
    - **Compile Class:** Compiles the source of the current class.

Preferences

Commands in the menu let you customize Jtest system settings.

- **Configuration Options:** Contains the following non-UI-related configuration options:
    - **Editor:** Opens a dialog box that lets you determine what editor is invoked when you view report files and edit your source. If the editor command includes white-space, enclose the command in quotation marks. To represent

the file parameter and the line number parameter, use the special tokens $FILE and $LINE in the lower text field.

- **Tips:** Contains the following options that configure context-sensitive tips:
  - **Reactivate All:** Reactivates all context-sensitive tips.
  - **Deactivate All:** Turns off all content-sensitive tips.
- **Report Format:** Contains options which let you determine whether Jtest's reports are formatted in HTML or in ASCII (text) format.
- **Report File:** Contains the following options that customize report file characteristics:
  - **Show All Classes Accessed:** Determines whether or not Jtest's single class reports annotate all sources for each class accessed during testing.
  - **Show Test Cases:** Determines whether or not Jtest includes test case information in single class reports.

- **UI Preferences:** Contains the following UI-related configuration options.
  - **Starting UI:** Determines whether the Class Testing UI or the Project Testing UI opens by default when Jtest is started. Choose **Last UI Visible** to have Jtest open the UI that was active the last time that you closed Jtest.
  - **Look and Feel:** Changes the look and feel of Jtest's UIs.
  - **Title Bar Background Color:** Determines the title bar's background color.
  - **Notion of Working:** Determines how the notion of working is represented.

- **Context Help Font:** Determines the size and type of the font used to display context-sensitive help text.

Tools

Commands in this menu access Jtest's tools.

- **Find Classes:** Starts the Find Classes UI that finds classes which can be tested by Jtest.
- **IDE Integration:** Enables you to integrate Jtest into third-party IDEs.
  - **IBM VisualAge 3.5:** Integrates Jtest into IBM VisualAge 3.5, 3.5.3, and 4.0. For more information on how Jtest works with VisualAge, see “Integrating VisualAge and Jtest” on page 154.
  - **Inprise JBuilder:** Integrates Jtest into JBuilder 4.0 or 5.0. For more information on how Jtest works with JBuilder, see “Integrating JBuilder and Jtest” on page 160.

Window

The command in this menu allows you to open the Project Testing UI.

- **Project Testing UI:** Opens the Project Testing UI (used to test a set of classes).

Help

Commands in this menu help you access additional information about Jtest.

- **Contents [F3]:** Opens the Jtest User's Guide.
- **Activate ContextHelp:** Activates context-sensitive help. After activating the help, move your cursor over the area on the UI that you would like to learn more about. A help window will open if that area has context-sensitive help.

- **Jtest API:** Opens the Jtest API documentation.
- **License:** Lets you enter or view your Jtest license.
- **Environment:** Contains the following commands that provide more information about the environment in which Jtest is running:
    - **Show CLASSPATH:** Displays the CLASSPATH that Jtest uses when it tests a class.
    - **Show User:** Display the name of the current Jtest user.
    - **Show OS:** Displays the operating system that Jtest is currently running on.
    - **Show Java:** Displays the Java version being used to run the Jtest UIs.
    - **Show JTEST_USER_DIR:** Displays the users directory that Jtest is using. (For example, C:\users\username\users or /users/username/users).
    - **Show Installation:** Displays the Jtest installation directory.
- **Support:** Allows you to choose from the following support options:
    - **Support Website:** Opens the Jtest support Web site.
    - **Live Help:** Opens a Web page from which you can receive live online help.
    - **Pack Support Files:** Automatically creates a zip file which can be sent to Jtest's Quality Consultants to help them answer your questions.
- **FAQ:** Opens the Jtest FAQ page.
- **Tutorial:** Opens the Jtest tutorial page.
- **Feedback:** Displays information about how to send feedback about Jtest to ParaSoft.
- **About:** Displays the Jtest version number and logo.

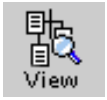
Class Testing UI Tool Bar

The following commands are available in the Jtest Class Testing UI tool bar.

Note: Buttons with a small downward arrow in their top right-corner have additional commands available in a shortcut menu. To access the shortcut menu containing additional commands, right-click the button.

Button	Name	Action
	**New Session**	Starts a new session by clearing any existing setup values or settings.
	**Project Testing UI**	Opens the Project Testing UI (used to test a set of classes).

	**Start All Tests**	Starts testing the class whose name appears in the **Class Name** field. Right-clicking this button displays the following commands in a shortcut menu: • **Start All Tests:** Starts testing the class. • **Static Analysis:** Starts running the selected type of Static Analysis tests. • **Dynamic Testing:** Starts running the selected type of Dynamic Analysis tests.
	**Stop**	Stops testing the class whose name appears in the **Class Name** field.
	**View Report**	Displays the Single Class Report for the current test. Right-clicking this button displays the following commands in a shortcut menu: • **View Report:** Displays the Single Class Report of the current test. • **Print ASCII Report:** Sends the report directly to the printer.

	**View Test Cases**	Opens the View Test Cases window (displays test cases that Jtest used for Dynamic Analysis).
	**View Class Metrics**	Displays class metrics.
	**Class Test Parameters**	Lets you view and edit the Class Test Parameters (parameters used for the current class test).
	**Global Test Parameters**	Lets you view and edit the Global Test Parameters. (parameters used for all Jtest tests).

	**Rules**	Displays nodes representing Jtest's built-in static analysis rules in the Global Test Parameters window. Right-clicking this button displays the following commands in a shortcut menu: • **Show Built-in Rules:** Displays nodes representing Jtest's built-in static analysis rules in the Global Test Parameters window. • **Show User-Defined Rules:** Displays nodes representing the rules you created with RuleWizard in the Global Test Parameters window. • **Show Rules Directory:** Displays the directory in which Jtest expects user-defined rules to be saved. • **Reload User-Defined Rules:** Prompts Jtest to check the Rules directory and refresh its list of user-defined rules. • **Launch RuleWizard:** Opens RuleWizard, the Jtest feature that lets you create your own rules and customize existing rules.

	**View Class Source**	Displays the source of the class currently under test. Right-clicking this button displays the following commands in a shortcut menu: • **View Class Source:** Displays the source of the current class in Jtest's source viewer. • **Edit Class Source:** Displays the source of the current class in your source editor. • **Locate .java file:** Displays the path to the .java file currently under test. • **Locate .class file:** Displays the path to the .class file currently under test. • **Compile Class:** Compiles the source of the current class.
	**Context Help**	Enables context-sensitive help. After clicking this button, move your cursor over the area on the UI that you would like to learn more about. A help window will open if that area has context-sensitive help.

Class Name Panel

This panel lets you specify what class you want Jtest to test.

To browse for the class to you want to test, click the **Browse** button, then use the file chooser to select the .class file that you want to test.

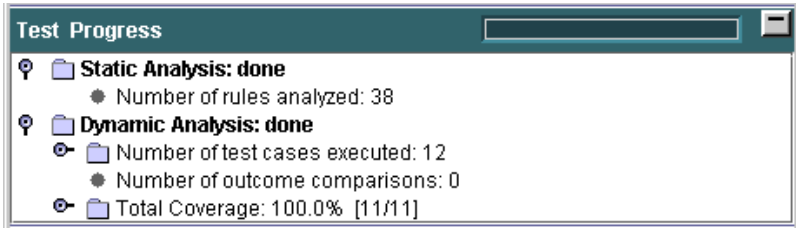
To enter a class directly, enter the fully qualified name of the class to test (without the .class extension) in the **Class Name** field.

Note: We recommend that you select the class to be tested using the **Browse** button. When you select a class using the **Browse** button, the working directory is set to the root directory of the class's package.

Test Progress Panel

The Test Progress panel is minimized by default. To view the information that it contains, you need to maximize it by clicking the Maximize button.

This panel displays the following test progress and coverage information:



- **Static Analysis:** Displays the progress of static analysis tests. While static analysis is being performed, a percentage indicating test progress is displayed to the right of this node. When a test is complete, the word “done” will appear to the right of this node

  The **Number of Rules Analyzed** node displays the number of static analysis rules analyzed.

- **Dynamic Analysis:** Displays the progress of dynamic analysis tests. While dynamic analysis is being performed, a percentage indicating test progress is displayed to the right of this node. When a test is complete, the word “done” will appear to the right of this node.

  Coverage information is shown only for classes on which Jtest has performed dynamic analysis. By default, dynamic analysis is only performed on the public classes; static analysis is performed on all classes found (public and non-public).

The **Number of Test Cases Executed** node displays the total number of test cases executed. These test cases are divided into two categories: automatic and user-defined. The **Automatic** node displays the number of automatically-generated test cases executed. The **User Defined** node displays the number of user-defined test cases executed.

The **Number of Outcome Comparisons** node displays the number of outcomes compared during black-box and regression testing.

The **Total Coverage** node displays the cumulative coverage that Jtest achieved. Jtest performs data coverage for the generated input categories; this means that the parts of the class that have been covered are thoroughly tested with respect to those inputs. The coverage reported is relative to the classes that have been accessed for the paths Jtest has tried. If some part of the class is not covered, it means that Jtest has not yet found a path leading to those statements or no path leads to those statements. In class testing mode, Jtest usually covers approximately 50% of a class's code. Sometimes Jtest will be able to test 100% of the class, and sometimes it will test less than 50% of the class.

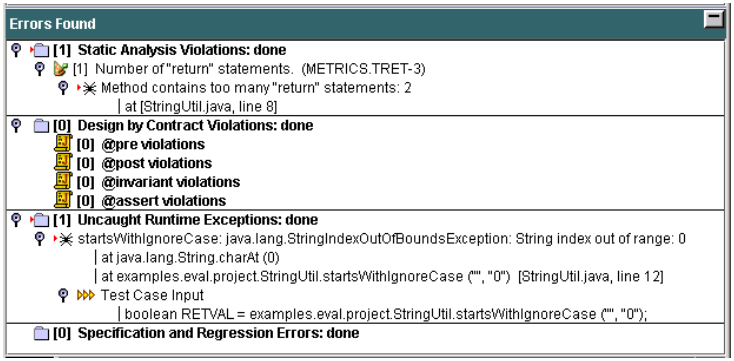
The **Total Coverage** branch's **Multi-condition branch** node displays coverage achieved on branches. A branch is a path of execution through the statements. Selection statements, such as "switch" and "if", have one or more branches per statement. Branch coverage is a measure of what percentage of branches were covered given the total number of branches in the code.

The **Total Coverage** branch's **Method** node displays coverage achieved on methods. Method coverage is a measure of what percentage of methods were covered given the total number of methods in the code.

The **Total Coverage** branch's **Constructor** node displays coverage achieved on constructors. Constructor coverage is a measure of what percentage of constructors were covered given the total number of constructors in the code.

Errors Found Panel

This panel displays information about the errors that Jtest found and lets you perform numerous actions that help you understand and customize results.



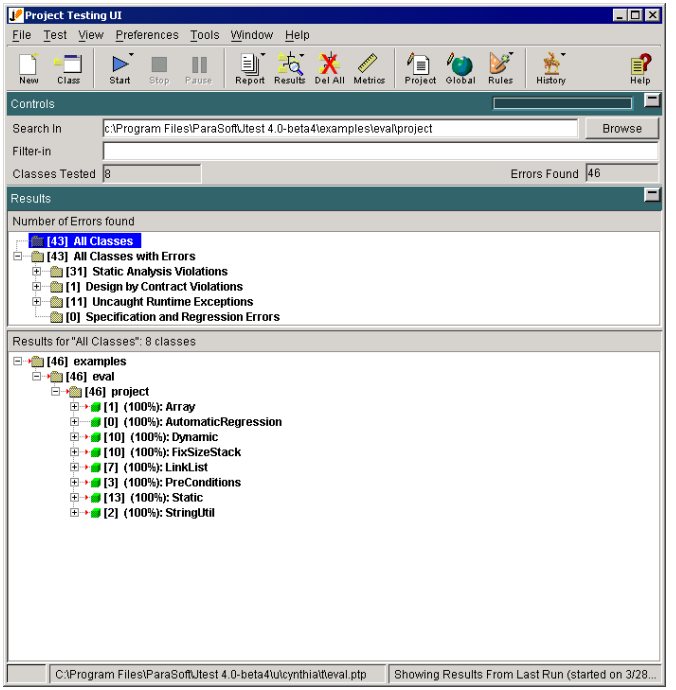
To learn more about this panel's branches and available options, see “Understanding the Errors Found Panel” on page 32 and “Exploring and Customizing Class Test Results” on page 37.

Project Testing UI

The Project Testing UI tests sets of classes. This UI consists of the following components:

- Menu Bar
- Tool Bar
- Controls Panel
- Results Panel

By default, the Class Testing UI opens when Jtest is started. To configure Jtest so that the Project Testing UI opens when Jtest is started, choose **Preferences> UI Preferences> Starting UI> Project Testing UI**.



For information on testing a set of classes in the Project Testing UI, see “Testing a Set of Classes” on page 40.

Project Testing UI Menu Bar

File

Commands in this menu control basic test functionality.

- **New:** Starts a new session by clearing any existing setup values or settings.
- **Open:** Opens an existing test specification saved as a .ptp file.
- **Open Recent:** Opens parameters of a recent test. Contains a list of the most recently opened projects; choose a file name from the list to open the associated parameters file. The **Clear List** command clears all items from this list.
- **Save:** Saves the current project test parameters in the test parameters file shown in the status bar.
- **Save As:** Saves the current project test parameters in the test parameters file that you specify.
- **Close UI:** Closes the Project Testing UI. If the Class Testing UI is not open, choosing this command will also close Jtest.
- **Exit:** Closes Jtest.

Test

Commands in this menu start, stop, and pause tests.

- **Start:** Starts testing the project specified in the **Search In** field.
- **Pause:** Temporarily the current test. Click this button again to resume testing.
  Note: Jtest will finish testing the current class before pausing.
- **Stop:** Stops the current test.

View

Commands in this menu display information related to the current test.

- **Report:** Contains the following report-related commands:
    - **View Report:** Displays the Project Report (contains project test parameters and details on all errors available in the Results panel).
    - **View Detail Report:** Displays the Detailed Project Report (contains project test parameters, class test parameters, and all information available in the Results panel).
    - **View Summary Report:** Displays the Summary Project Report (contains one line for each error available in the Results panel).
- **Delete All:** Removes all of the results in the lower Results window.
- **Metrics:** Displays project and average class metrics.
- **Project Test Parameters:** Lets you view and edit the current Project Test Parameters (parameters used for the current project test).
- **Global Test Parameters:** Lets you view and edit the Global Test Parameters (parameters used for all Jtest tests).
- **History:** Displays a record of all the runs for this Project test

Preferences

Commands in the menu let you customize Jtest system settings.

- **Configuration Options:** Contains the following non-UI-related configuration options:
    - **Editor:** Opens a dialog box that lets you determine what editor is invoked when you view report files and edit your source. If the editor command includes white-space, enclose the command in quotation marks. To represent the file parameter and the line number parameter, use

the special tokens $FILE and $LINE in the lower text field.

- **Tips:** Contains the following options that configure context-sensitive tips:
  - **Reactivate All:** Reactivates all context-sensitive tips.
  - **Deactivate All:** Turns off all content-sensitive tips.
- **Report Format:** Contains options which let you determine whether Jtest's reports are formatted in HTML or in ASCII (text) format.
- **Report File:** Contains the following options that customize report file characteristics:
  - **Show All Classes Accessed:** Determines whether or not Jtest's single class reports annotate all sources for each class accessed during testing.
  - **Show Test Cases:** Determines whether or not Jtest includes test case information in single class reports.
- **Default Results Loading:** Determines what results are loaded when a .ptp file is opened.
  - **Last:** Only results from the most recent test will be loaded when a .ptp file is opened.
  - **All:** All previous results (including results from the most recent test run) will be loaded when a .ptp file is opened.
  - **None:** Prevents any results from being loaded when a .ptp file is opened.
- **Report Type:** Determines whether Jtest's reports are formatted in HTML or in ASCII.

- **UI Preferences:** Contains the following UI-related configuration options.

- **Starting UI:** Determines whether the Class Testing UI or the Project Testing UI opens by default when Jtest is started. Choose **Last UI Visible** to have Jtest open the UI that was active the last time that you closed Jtest.
- **Look and Feel:** Changes the look and feel of Jtest's UIs.
- **Title Bar Background Color:** Determines the title bar's background color.
- **Notion of Working:** Determines how the notion of working is represented.
- **Context Help Font:** Determines the size and type of the font used to display context-sensitive help text.

Tools

Commands in this menu access Jtest's tools.

- **IDE Integration:** Enables you to integrate Jtest into third-party IDEs.
  - **IBM VisualAge:** Integrates Jtest into IBM VisualAge 3.5, 3.5.3, and 4.0. For more information on how Jtest works with VisualAge, see “Integrating VisualAge and Jtest” on page 154.
  - **Inprise JBuilder:** Integrates Jtest into JBuilder 4.0 or 5.0. For more information on how Jtest works with JBuilder, see “Integrating JBuilder and Jtest” on page 160

Window

- **Class Testing UI:** Opens the Class Testing UI (used to test a single class or view results for a single class).

Help

Commands in this menu help you access additional information about Jtest.

- **Contents [F3]:** Opens the Jtest User's Guide.
- **Activate ContextHelp:** Activates context-sensitive help. After activating the help, move your cursor over the area on the UI that you would like to learn more about. A help window will open if that area has context-sensitive help.
- **Jtest API:** Opens the Jtest API documentation.
- **License:** Lets you enter or view your Jtest license.
- **Environment:** Contains the following commands that provide more information about the environment in which Jtest is running:
    - **Show CLASSPATH:** Displays the CLASSPATH that Jtest uses when it tests a class.
    - **Show User:** Display the name of the current Jtest user.
    - **Show OS:** Displays the operating system that Jtest is currently running on.
    - **Show Java:** Displays the Java version being used to run the Jtest UIs.
    - **Show JTEST_USER_DIR:** Displays the users directory that Jtest is using. (For example, C:\users\user-name\users or /users/username/users).
    - **Show Installation:** Displays the Jtest installation directory.
- **Support:** Allows you to choose from the following support options:
    - **Support Website:** Opens the Jtest support Web site.
    - **Live Help:** Opens a Web page from which you can receive live online help.
    - **Pack Support Files:** Automatically creates a zip file which can be sent to Jtest's Quality Consultants to help them answer your questions.
- **FAQ:** Opens the Jtest FAQ page.

- **Tutorial:** Opens the Jtest tutorial page.
- **Feedback:** Displays information about how to send feedback about Jtest to ParaSoft.
- **About:** Displays the Jtest version number and logo.

Project Testing UI Tool Bar

The following commands are available in the Project UI tool bar.

Note: Buttons with a small downward arrow in their top right-corner have additional commands available in a shortcut menu. To access the shortcut menu containing additional commands, right-click the button.

Button	Name	Action
	**New Session**	Starts a new session by clearing any existing setup values or settings.
	**Class Testing UI**	Opens the Class Testing UI (used to test a single class or view results for a single class).

	**Start finding and testing classes**	Starts finding and testing classes in the area specified in the **Search In** parameter. Right-clicking this button displays the following commands in a shortcut menu: • **Start All Tests:** Starts testing the class. • **Static Analysis:** Starts running the selected type of Static Analysis tests. • **Dynamic Testing:** Starts running the selected type of Dynamic Analysis tests. **Note:** Jtest will not test a previously tested class unless that class was modified since the last test.
	**Stop**	Stops finding and testing classes.
	**Pause**	Temporarily stops finding and testing classes. Click this button again to resume testing. **Note:** Jtest will finish testing the current class before pausing.

	**View Report**	Displays a report of the current project test. The report contains only the classes and errors that are contained in the Results panel at the time this button is clicked. To limit the classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **Report** button. Right-clicking this button displays the following commands in a shortcut menu: • **View Report:** Displays the Project Report (contains project test parameters and details on all errors available in the Results panel). • **View Detail Report:** Displays the Detailed Project Report (contains project test parameters, class test parameters, and all information available in the Results panel). • **View Summary Report:** Displays the Summary Project Report (contains one line for each error available in the Results panel).

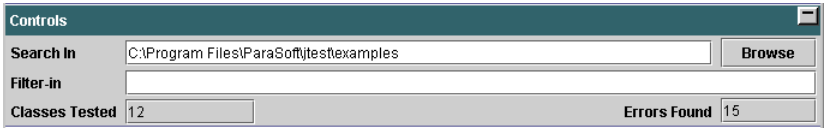
	**View All Results**	Displays all results. Right-clicking this button displays the following commands in a shortcut menu: • **View All Results:** Displays all results. • **View Results From Last Run:** Displays only results from the last run.
	**Delete All**	Removes all of the results in the lower Results window.
	**Metrics**	Displays project and average class metrics.
	**Project Test Parameters**	Lets you view and edit the current Project Test Parameters (parameters used for the current project test).

	**Rules**	Displays nodes representing Jtest's built-in static analysis rules in the Global Test Parameters window. Right-clicking this button displays the following commands in a shortcut menu: • **Show Built-in Rules:** Displays nodes representing Jtest's built-in static analysis rules in the Global Test Parameters window. • **Show User-Defined Rules:** Displays nodes representing the rules you created with RuleWizard in the Global Test Parameters window. • **Show Rules Directory:** Displays the directory in which Jtest expects user-defined rules to be saved. • **Reload User-Defined Rules:** Prompts Jtest to check the Rules directory and refresh its list of user-defined rules. • **Launch RuleWizard:** Opens RuleWizard, the Jtest feature that lets you create your own rules and customize existing rules.

	**Global Test Parameters**	Lets you view and edit the Global Test Parameters (parameters used for all Jtest tests).
	**Test History**	Displays a record of all the runs for this Project test, or all Project tests. Right-clicking this button displays the following commands in a shortcut menu: • **Test History:** Displays a record of all runs of the current test. • **Global History:** Displays a record of all Project tests.
	**Context Help**	Enables context-sensitive help. After clicking this button, move your cursor over the area on the UI that you would like to learn more about. A help window will open if that area has context-sensitive help.

Controls Panel

This panel lets you specify the fundamental parameters used during a project test and reports basic data about a project test.



You can enter two parameters in this panel:

- **Search In:** Specifies where Jtest should start searching for classes to test. The parameter can be a directory, a jar file, a zip file, or a .class file

  If the parameter is a directory, Jtest will recursively traverse the path's subdirectories, zip files, and jar files, searching for and testing any classes it finds.

  If the parameter is a jar or zip file, Jtest will open the file and search it for classes in which to find errors.

  To browse for the directory, jar file, or zip file that you want Jtest to start searching and testing, click the **Browse** button, locate and select the desired directory, jar file, or zip file in the file chooser, then click **Open**.

- **Filter-in:** Tells Jtest to find and test only classes that match the given regular expression. This regular expression works like the file-matching utility of a Unix shell.

  To test only classes with the string XYZ in the class name use:

XYZ

To test only classes with names end with XYZ use:
*XYZ

To test only classes in the packages com.util or com.lib use:
{com.util.*,com.lib.*}

For example, if you want Jtest to look only for classes in the DB package, use
`DB.*`

When this field is left empty, all classes found will be tested.

The following table describes the difference between perl's regular expressions and file matching:

FileRegex	Regex
*	.*
.	\.
{	(?:
{?!	(?!
{?=	(?=
}	)
?	.
{,}	(|)
*.java	.*\.java$

*.{java,html}	.*\.(java|html) $

For a reference on Regular Expressions, see

http://www.perl.com/pub/doc/manual/html/pod/perlre.html

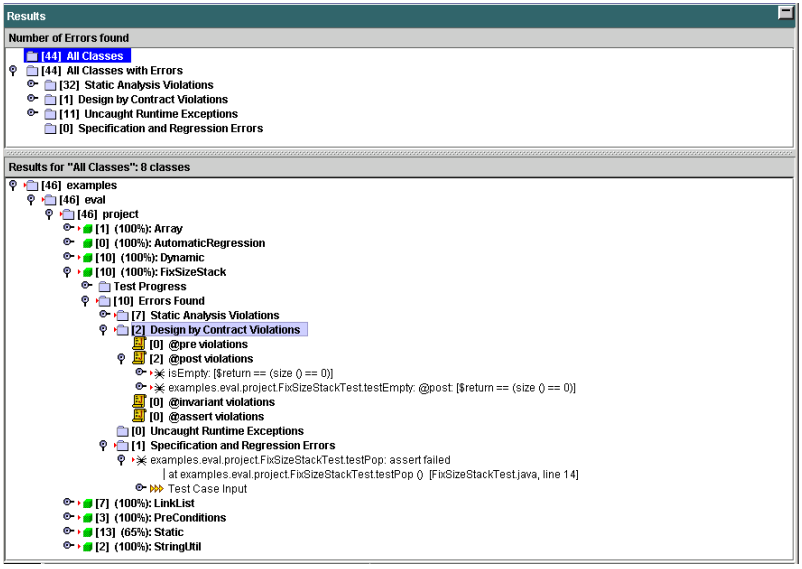
If you want to use regular expressions instead of File regexpressions, change the jtest.properties file's COM.parasoft.util.Regexp.Type value to 2 instead of 1. This file is located at:
Windows OS: <jtest install dir>/u/<user name>/jtest.properties
Unix OS: $HOME/.jtest/jtest.properties

Two test result parameters are displayed in this panel:

- **Classes Tested:** Displays the number of classes tested by Jtest.
- **Errors Found:** Displays the number of errors found by Jtest.

Project Testing UI Results Panel

This panel displays information about the errors that Jtest found during a project test. It also lets you perform numerous actions that help you understand and customize results.



To learn more about this panel's branches and available options, see “Understanding the Results Panel” on page 45 and “Exploring and Customizing Project Test Results” on page 53.

Global Test Parameters

This window lets you view and edit the Global Test Parameters used throughout Jtest. To open this window, click the **Global** button in either the Class Testing UI or Project Testing UI.

Descriptions of Global Test Parameters tree branches are divided into three categories:

- Global Test Parameters - Static Analysis
- Global Test Parameters - Dynamic Analysis
- Global Test Parameters - Common Parameters

CLASSPATH Requirements

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

- The '.class' files for the classes you want to test must be available. A '.class' file is a compiled Java source. Without a '.class' 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 referenced by the tested '.class' files must be available to Jtest. This is done by adding their location to the CLASSPATH.
- If the '.class' files are in directories, '.zip' files, or '.jar' files, the '.class' files must be accessible by Jtest.

If during testing, Jtest finds ClassNotFoundExceptions or NoClassDefFoundErrors, or if it reports that it could not find the package on "imports", the CLASSPATH is not set properly. If this occurs, you need to set the system CLASSPATH variable to include every class referenced (recursively) by the tested class prior to testing. Check that the CLASSPATH includes the parent directory of the directory hierarchy. For example, if you are testing com.company.MyClass and Jtest reports that it could not

find a package referenced by MyClass, it is probably because the 'com' directory is not on the CLASSPATH.

You can override the CLASSPATH environment variable in the Global Test Parameters, the Class Test Parameters, or the Project Test Parameters.

Global Test Parameters - Static Analysis

Static Analysis

Contains parameters that control static analysis.

Perform Static Analysis

Flag that controls whether or not static analysis is performed when a class is tested.

Note: This flag appears in all parameter levels.

Rules

Contains rules that can be applied when Jtest performs static analysis.

Severity Levels Enabled

Contains severity level flags. Each rule has a severity level associated with it. A rule is enforced only if both the rule and its severity level are enabled. This branch controls which severity levels are enabled.

To enable all severity level, right-click this node, then choose **Enable All** from the shortcut menu.

To view the number of active rules, right-click this node, then choose **Show Number Active Rules** from the shortcut menu.

Level 1-5

Flags that control whether or not the rules of a particular severity level are applied.

Built-in Rules

Contains built-in rules shipped with Jtest. A specific rule can be

enabled or disabled using the flag associated with that rule. A rule is enforced only if both the rule and its severity level are enabled.

Possible Bugs

Contains rules that check for possible bugs in the code (i.e., the code compiles, but the programmer made some typos while entering the code).

Object-Oriented Programming

Contains rules that enforce good Object-Oriented Programming practices.

Unused Code

Contains rules that check for unused code.

Initialization

Contains rules that enforce the explicit initialization of the variables.

Coding Standards

Contains rules that enforce good programming practices.

Naming Conventions

Contains rules that enforce common naming conventions.

Javadoc Comments

Contains rules related to Javadoc comments.

Portability

Contains rules related to portability.

Optimization

Contains rules that check for non-optimal constructs.

Garbage Collection

Contains rules related to garbage collection.

Threads and Synchronization

Contains rules related to threads and synchronization.

Enterprise JavaBeans

Contains rules related to Enterprise JavaBeans (EJB).

Class Metrics

Contains rules that measure class and method metrics. Lets you modify upper and lower thresholds for each metric (for information on modifying thresholds, see “Customizing Class Metrics” on page 82.

Project Metrics

Contains rules that measure project and average class metrics.

Miscellaneous

Contains miscellaneous rules.

Design by Contract

Contains rules that enforce proper Design by Contract contract formation.

Internationalization

Contains rules that facilitate code internationalization.

Security

Contains rules related to security.

Servlets

Contains rules related to servlets.

Global Static Analysis

Contains rules that perform global static analysis.

Note: These rules are only checked when you test a project in the Project Testing UI.

User Defined Rules

Contains user defined rules that were created in RuleWizard. Specific rules can be enabled or disabled using the flag associated with each particular rule (listed by category). A rule is enforced only if both the rule and its severity level are enabled.

Global Test Parameters - Dynamic Analysis

Dynamic Analysis

Contains parameters that control dynamic analysis.

Perform Dynamic Analysis

Flag that controls whether or not dynamic analysis is performed every time a class is tested.

Note: This flag appears in all parameter levels.

Test Case Generation

Contains parameters that control test case generation.

Automatic

Contains parameters that control the generation of automatic test cases.

Test calling sequences up to length

By default, Jtest tests each method by calling it independently and generating arguments to it. That is, Jtest basically tries calling sequences of length 1.

This option can be used to tell Jtest to try calling sequences longer than 1. If a calling sequence of length N is specified, Jtest will first try all calling sequences of length 1, then all calling sequences of length 2, and so on.

Note: Jtest will attempt to show errors with the shortest calling sequences that can cause the errors. Most errors should have a calling sequences of length 1 or 2.

Test Methods

Contains flags that control which methods can be called directly in the calling sequence generated by Jtest

Jtest will only directly call the methods whose accessibility is selected here.

Note that the methods that are not called directly are still tested indirectly, through calls to the methods that are called directly.

public

Flag that controls if Jtest tests all of the class's public methods.

protected

Flag that controls if Jtest tests all of the class's protected methods.

package-private of package-private classes

Flag that controls if Jtest tests all package-private methods in package-private classes.

A package-private method is a method without any accessibility qualifier (e.g., public, protected, or private). package-private methods are only accessible by classes within the same package as the method.

A package-private class is a class without the "public" accessibility qualifier. package-private classes are only accessible by other classes within the same package as the method.

package-private of public classes

Flag that controls if Jtest tests all package-private methods in public classes.

A package-private method is a method without any accessibility qualifier (e.g., public, protected, or private). package-private classes are only accessible by classes within the same package as the method.

private

Flag that controls if private methods are called directly.

Common

Contains parameters shared by both automatic and user-defined test case generation.

Static Global Initialization

The code associated with this node (as well as the code associated with the Static Project Initialization and Static Class Initialization nodes) is executed before any test case is executed, and can be used to setup and initialize the class if needed. You can invoke only static methods from these initialization nodes. See “Setting an Object to a Certain State” on page 106 for more information.

Inputs Repository

Stores input values that can later be added to a method argument node.

Note: This feature is deprecated. For more information about defining inputs for test cases, see “Adding Method Inputs” on page 119.

Test Case Execution

Contains parameters that control test case execution.

Execute Automatic

Flag that controls whether or not automatic test cases are executed every time a class is tested.

Note: This flag appears in all parameter levels.

Execute User Defined

Flag that controls whether or not user-defined test cases are executed every time a class is tested.

Note: This flag appears in all parameter levels.

Stubs

Contains stub-related options.

For more information on stubs, see “Testing Classes That Reference External Resources” on page 93 and “Using Custom Stubs” on page 98.

Options for Automatic Test Cases

Contains options that let you control what type of stubs are used while running the automatically-generated test cases. You can choose **Use Automatic Stubs**, **Use User Defined Stubs**, or neither. Jtest will call the actual external method if neither of these stub types are selected, or if **Use User Defined Stubs** is selected, but no stubs are defined for a particular method reference.

Use Automatic Stubs (white-box stubs)

If selected, Jtest will automatically generate stubs for external resources while running the automatically-generated test cases. For more information on Automatic Stubs, see “Testing Classes That Reference External Resources” on page 93.

Use User Defined Stubs

If selected, Jtest will use user-defined stubs when the class under test references external resources. For more information on User Defined Stubs, see “Testing Classes That Reference External Resources” on page 93 and “Using Custom Stubs” on page 98.

Options for User Defined Test Cases

Contains options that let you control what type of stubs are used while running the user defined test cases. You can choose **Use User Defined Stubs**, or you can leave this option unselected. Jtest will call the actual method if this option is not selected, or if this option is selected, but no user defined stubs are defined for a particular method reference.

Use User Defined Stubs

If selected, Jtest will use user-defined stubs when the class under test references external resources. For more information on User Defined

Stubs, see “Testing Classes That Reference External Resources” on page 93 and “Using Custom Stubs” on page 98.

Note: This flag appears in all parameter levels.

“Tested Set” Includes

Defines the “Tested Set”: the set of classes and methods included in the current test. When a class or method in the Tested Set references a class or method that is inside that Tested Set, the actual class or method is accessed. When a class or method in the Tested Set references a class or method that is outside that Tested Set, stubs are called.

For more information about Tested Set, see “Defining Which Classes are “External”” on page 94.

Pre-filtering Suppression Categories

Contains suppression categories that can be applied when the test cases are executed.

Exceptions in Throws Clause

If selected, Jtest will not report exceptions occurring in methods that are declared with the exception's type in the throws clause of the method.

Note: This flag appears in all parameter levels.

Direct IllegalArgumentExceptions

If selected, Jtest will not report IllegalArgumentExceptions that are thrown directly by a throw statement.

Note: This flag appears in all parameter levels.

Explicitly Thrown Exceptions

If selected, Jtest will not report exceptions that are explicitly thrown by user code with a throw statement.

Note: This flag appears in all parameter levels.

Exceptions Caught By Empty Catch

If selected, Jtest will not report exceptions caught by an empty catch block.

Note: This flag appears in all parameter levels.

Direct NullPointerExceptions

If selected, Jtest will not report exceptions that can occur because a null object is passed to a method which subsequently dereferences the object, thus causing the NullPointerException.

Note: This flag appears in all parameter levels.

Automatically Instrument "Design by Contract" Comments

Determines whether or not Jtest automatically instruments Design by Contract comments as classes are loaded into Jtest.

If you want Jtest to use the information in a class's "Design by Contract" javadoc comments (e.g., to automatically create test cases that verify functionality, or to use these comments to suppress certain exceptions), it is necessary to instrument these comments.

If you never use Design by Contract comments, you can prevent Jtest from instrumenting classes by disabling this option.

Note: This flag appears in all parameter levels.

Test Case Evaluation

Contains parameters that control test case evaluation.

Report Uncaught Runtime Exceptions

Flag that controls whether or not Jtest reports uncaught runtime exceptions that occur in the tested class.

Note: This flag appears in all parameter levels.

Perform Automatic Regression Testing

Flag that controls whether or not Jtest performs Automatic Regression Testing for the tested class.

Note: This flag appears in all parameter levels.

Suppressions Table

Double-clicking this leaf invokes the dynamic analysis Suppression Table which lets you suppress dynamic analysis violations.

Global Test Parameters - Common Parameters

Common Parameters

Contains parameters shared by both static and dynamic analysis.

Directories

Contains parameters related to directories.

Working Directory

Determines the directory that is used as the current working directory when testing a class.

This directory will be used as "." on the CLASSPATH.

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, Jtest uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

The following tokens are treated specially:

- $PARENT: This token is replaced by the parent parameter value.
- $PARAMS_DIR: This token is replaced by the directory that includes the parameters directory.
- $INSTALL_DIR: This token is replaced by the Jtest installation directory.
- $NAME: This token is replaced by the value of the environment variable NAME.

The actual value that will be used is shown in parentheses.

Results

Determines where the test results will be stored.

The following tokens are treated specially:

- $DEFAULT: In project tests, this token is replaced by a path relative to the location of the project test parameters (.ptp) file. This token only applies to project tests.
- $PARENT: This token is replaced by the parent parameter value.
- $PARAMS_DIR: This token is replaced by the directory that includes the parameters directory.
- $INSTALL_DIR: This token is replaced by the Jtest installation directory.
- $NAME: This token is replaced by the value of the environment variable NAME.

The actual value that will be used is shown in parentheses.

javac/javac-like Parameters

Contains parameters equivalent to parameters used in java or javac.

-classpath

Overrides the CLASSPATH environment variable with the list of entries specified here (an entry is a directory, zip file, or jar file).

This option is equivalent to the java interpreter's -classpath flag.

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, Jtest uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

The following tokens are treated specially:

- $PARENT: This token is replaced by the parent parameter value.
- $PARAMS_DIR: This token is replaced by the directory that includes the parameters directory.

- $INSTALL_DIR: This token is replaced by the Jtest installation directory.
- $NAME: This token is replaced by the value of the environment variable NAME.

The actual value that will be used is shown in parentheses.

-cp

Prepends the CLASSPATH environment variable with the list of entries specified here (an entry is a directory, zip file, or jar file).

This option is equivalent to the JRE's -cp flag.

The token $PARENT receives special treatment and is replaced by the parent parameter value.

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, Jtest uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

The following tokens are treated specially:

- $PARENT: This token is replaced by the parent parameter value.
- $PARAMS_DIR: This token is replaced by the directory that includes the parameters directory.
- $INSTALL_DIR: This token is replaced by the Jtest installation directory.
- $NAME: This token is replaced by the value of the environment variable NAME.

The actual value that will be used is shown in parentheses.

System Properties

Defines system properties. This parameter is equivalent to the -D flag of the Java interpreter and is used to define properties for the class

being tested.

System properties are defined by naming the property and assigning a value to the property. Use a space to separate properties if multiple properties are defined.

Example: `property.one=On PROPERTY_TWO=d:/temp`

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, Jtest uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

The following tokens are treated specially:

- $PARENT: This token is replaced by the parent parameter value.
- $PARAMS_DIR: This token is replaced by the directory that includes the parameters directory.
- $INSTALL_DIR: This token is replaced by the Jtest installation directory.
- $NAME: This token is replaced by the value of the environment variable NAME.

The actual value that will be used is shown in parentheses.

-Xbootclasspath

Overrides location of bootstrap class files.

If Class is a Test Class

Determines how Jtest behaves when the class under test is a Test Class.

Run the tests defined in the Class

If selected, Jtest runs the tests defined in the Test Class and Jtest will *not* test the Test Class itself (Jtest will not perform static analysis on the class or create test cases for it). If you select this option, you can-

not select the **Test the Test Class itself** option.

Test the Test Class itself

If selected, Jtest tests the Test Class as it would test any other class. Jtest *will* test perform static analysis on the class and create test cases for it). If you select this option, you cannot select the **Run the tests defined in the Class** option.

Source Path

Determines where Jtest looks for the source of a class.

Path to JDK Directory

Specifies the path to the JDK installation directory. Jtest only uses this JDK installation to compile classes; it runs classes with the JRE that is shipped with Jtest.

Class Test Parameters

This window lets you view and edit parameters that are specific to a certain class.

In the Class Testing UI, you can open this window by clicking the **Class** button.

You can also open this window from the Project Testing UI Right-click the Result panel node whose name corresponds to the class whose parameters you want to modify, then choose **Edit Class Test Parameters** from the shortcut menu.

Descriptions of Class Test Parameters tree branches are divided into three categories:

- Class Test Parameters - Static Analysis
- Class Test Parameters - Dynamic Analysis
- Class Test Parameters - Common Parameters

CLASSPATH Requirements

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

- The '.class' files for the classes you want to test must be available. A '.class' file is a compiled Java source. Without a '.class' 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 referenced by the tested '.class' files must be available to Jtest. This is done by adding their location to the CLASSPATH.
- If the '.class' files are in directories, '.zip' files, or '.jar' files, the '.class' files must be accessible by Jtest.

If during testing, Jtest finds ClassNotFoundExceptions or NoClassDefFoundErrors, or if it reports that it could not find the package on "imports",

the CLASSPATH is not set properly. If this occurs, you need to set the system CLASSPATH variable to include every class referenced (recursively) by the tested class prior to testing. Check that the CLASSPATH includes the parent directory of the directory hierarchy. For example, if you are testing com.company.MyClass and Jtest reports that it could not find a package referenced by MyClass, it is probably because the 'com' directory is not on the CLASSPATH.

You can override the CLASSPATH environment variable in the Global Test Parameters, the Class Test Parameters, or the Project Test Parameters.

Class Test Parameters - Static Analysis

Static Analysis

See description in Global Test Parameters.

Perform Static Analysis

See description in Global Test Parameters.

Rules

See description in Global Test Parameters.

Severity Levels

See description in Global Test Parameters.

Level 1-5

See description in Global Test Parameters.

Suppressed Messages

Contains the list of specific static analysis messages that have been suppressed for this class.

Class Test Parameters - Dynamic Analysis

Dynamic Analysis

See description in Global Test Parameters.

Perform Dynamic Analysis

See description in Global Test Parameters.

Test Case Generation

See description in Global Test Parameters.

Automatic

See description in Global Test Parameters.

Test calling sequences up to length

See description in Global Test Parameters.

Test Methods

See description in Global Test Parameters.

public

Flag that controls if public methods are called directly.

protected

Flag that controls if protected methods are called directly.

package-private

Flag that controls if package-private methods are called directly.

private

Flag that controls if private methods are called directly.

Restricted Inputs

By default, Jtest will try to generate any input for the methods of the class. Use these nodes to restrict the inputs that Jtest will generate.

"THIS" object

Specifies what value Jtest will use by default when testing instance methods of the given class. Right-clicking this node displays a shortcut menu that allows you to set restricted inputs, add inputs from local repository, or add inputs from the global repository. This shortcut menu contains the following options:

- **Set Restricted Input:** Lets you add a valid Java expression as a simple input value. If you reference classes that are not in the same package as the tested class, make sure to add import statements for these classes. For information about adding imports, see “Specifying Imports” on page 132.
- **Add From Local Repository:** Contains menu items associated with the values available in the local repositories.
  Choose an input's menu item to add that input to the node.
  You can add inputs to the local repository in **Class Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.
- **Add From Global Repository:** Contains menu items associated with the values available in the global repository.
  Choose a menu item to add the input to the node.
  You can add inputs to the global repository in **Global Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.

User Defined

Contains parameters that control the generation of the user defined test cases. [n]= number of test cases defined.

Method Inputs

Contains nodes that can be used to specify the set of inputs with which you want Jtest to test the class. [n]= number of test cases.

[Method name]

Use these nodes to specify the inputs to be used for the named method. [n]= number of test cases for this method.

[Argument name]

Use the associated shortcut menu to add valid Java expressions as input values to this argument. [n]= number of inputs for this argument.

Shortcut menu commands available include:

- **Add Input Value:** Lets you add a simple input value.
- **Add From Local Repository:** Contains menu items associated with the values available in the local repositories.

  Choose an input's menu item to add that input to the node.

  You can add inputs to the local repository in **Class Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.
- **Add From Global Repository:** Contains menu items associated with the values available in the global repository.

  Choose an input menu item to add the input to the node.

  You can add inputs to the global repository in **Global Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.
- **Delete All Inputs:** Removes all existing inputs.

Test Classes

Test classes let you add test cases that are too complex or difficult to be added as method inputs. A test class is a class that extends

jtest.TestClass and is used to specify test cases that Jtest should use to test the class. You can write your own test class, or use your JUnit classes. For information on adding Test Classes, see “Adding Test Classes” on page 125.

[n]= Total number of test cases defined by all of the test classes.

Common

Contains parameters shared by both automatic and user-defined test case generation.

Imports

Contains imports shared by all the code used in the specification. See “Specifying Imports” on page 132 for more information.

Static Class Initialization

See description in Global Test Parameters.

Inputs Repository

See description in Global Test Parameters

Test Case Execution

See description in Global Test Parameters.

Execute Automatic

See description in Global Test Parameters.

Execute User Defined

See description in Global Test Parameters.

Stubs

See description in Global Test Parameters.

Options for Automatic Test Cases

See description in Global Test Parameters.

Use Automatic Stubs (white-box stubs)

See description in Global Test Parameters.

Use User Defined Stubs

See description in Global Test Parameters.

Options for User Defined Test Cases

See description in Global Test Parameters.

Use User Defined Stubs

See description in Global Test Parameters.

“Tested Set” Includes

See description in Global Test Parameters.

Stubs Class

Indicates what stub class to use while testing this class. If you use the token $DEFAULT, Jtest will automatically search for and use a class named (class_under_test_name)Stubs that extends jtest.Stubs. To enter the specific location of the appropriate stubs class, right-click this option, choose **Edit** from the shortcut menu, then enter the path to the stubs class.

For more information on stubs, see “Testing Classes That Reference External Resources” on page 93 and “Using Custom Stubs” on page 98.

Pre-filtering Suppression Categories

See description in Global Test Parameters.

Exceptions in Throws Clause

See description in Global Test Parameters.

DirectIllegalArgumentExceptions

See description in Global Test Parameters.

Explicitly Thrown Exceptions

See description in Global Test Parameters.

Exceptions Caught By Empty Catch

See description in Global Test Parameters.

DirectNullPointerExceptions

See description in Global Test Parameters.

Automatically Instrument “Design by Contract” Comments

See description in Global Test Parameters.

Test Case Evaluation

Contains parameters that control test case evaluation. For more information about test case evaluation, see “Viewing and Validating Test Cases” on page 171.

Report Uncaught Runtime Exceptions

See description in Global Test Parameters.

Perform Automatic Regression Testing

See description in Global Test Parameters.

Specification and Regression Test Cases

These test cases are used as reference test cases when Jtest performs regression and black-box testing. When tests are run, the outcomes for the run are compared with these outcomes. If a discrepancy exists, an error is reported.

Jtest automatically adds the automatic test cases that increase coverage to this list. All user-defined test cases are added.

Shortcut menus let you specify whether outcomes are correct or incorrect.

If you change specification and regression test cases and want to restore the set used during the actual tests, right-click the **Specification and Regression Test Cases** node, then choose the **Reload** option from the shortcut menu. Jtest will then reload the original test cases.

[method name]

Contains test cases for this method.

Test Cases

Contains all the information for a test case.

Test Case Input

Input that defines the test case.

The input for automatic test cases is the calling sequence.

Outcomes

Outcomes for this test case. Verify if the outcomes are correct or incorrect according to the class specification and set their state using the shortcut menus.

When the outcome is an object, Jtest automatically chooses the toString method to show its state.

If a method named jtestInspector is defined for the object's class, Jtest will only use the return value of this method to show the object state.

If no toString or jtestInspector methods are defined, Jtest will heuristically choose some public instance methods for that object to show its state.

If the method under test is a static method, Jtest will heuristically choose public static methods to show the class state. If the methods Jtest chose are not enough, declare a static method called sjtestInspector for the class. Jtest will use the return value of this method to

show the object class.

[n]= number of outcomes for this test case.

Exceptions

Indicates whether an exception occurred, and, if so, what type of exception occurred.

Class Test Parameters - Common Parameters

Common Parameters

See description in Global Test Parameters.

Directories

See description in Global Test Parameters.

Working Directory

See description in Global Test Parameters.

Results

See description in Global Test Parameters.

javac/javac-like Parameters

See description in Global Test Parameters.

-classpath

See description in Global Test Parameters.

-cp

See description in Global Test Parameters.

System Properties

See description in Global Test Parameters.

-Xbootclasspath

See description in Global Test Parameters.

If Class is a Test Class

See description in Global Test Parameters.

Run the tests defined in the Class

See description in Global Test Parameters.

Test the Test Class itself

See description in Global Test Parameters.

Source Path

See description in Global Test Parameters

Project Test Parameters

This window lets you view and edit parameters that apply to the current project test. To open this window, click the **Project** button in the Project Testing UI tool bar.

Descriptions of Project Test Parameters tree branches are divided into three categories:

- Project Test Parameters - Static Analysis
- Project Test Parameters - Dynamic Analysis
- Project Test Parameters - Common Parameters, Search Parameters, Classes in Project

CLASSPATH Requirements

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

- The '.class' files for the classes you want to test must be available. A '.class' file is a compiled Java source. Without a '.class' 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 referenced by the tested '.class' files must be available to Jtest. This is done by adding their location to the CLASSPATH.
- If the '.class' files are in directories, '.zip' files, or '.jar' files, the '.class' files must be accessible by Jtest.

If during testing, Jtest finds ClassNotFoundExceptions or NoClassDefFoundErrors, or if it reports that it could not find the package on "imports", the CLASSPATH is not set properly. If this occurs, you need to set the system CLASSPATH variable to include every class referenced (recursively) by the tested class prior to testing. Check that the CLASSPATH includes the parent directory of the directory hierarchy. For example, if you are testing com.company.MyClass and Jtest reports that it could not

find a package referenced by MyClass, it is probably because the 'com' directory is not on the CLASSPATH.

You can override the CLASSPATH environment variable in the Global Test Parameters, the Class Test Parameters, or the Project Test Parameters.

Project Test Parameters - Static Analysis

Static Analysis

See description in Global Test Parameters.

Perform Static Analysis

See description in Global Test Parameters.

Rules

See description in Global Test Parameters.

Severity Levels Enabled

See description in Global Test Parameters.

Level 1-5

See description in Global Test Parameters.

Project Test Parameters - Dynamic Analysis

Dynamic Analysis

See description in Global Test Parameters.

Perform Dynamic Analysis

See description in Global Test Parameters.

Test Case Generation

See description in Global Test Parameters.

Automatic

See description in Global Test Parameters.

Test calling sequences up to length

See description in Global Test Parameters.

Test Methods

See description in Global Test Parameters.

public

See description in Global Test Parameters.

protected

See description in Global Test Parameters.

package-private of package-private classes

See description in Global Test Parameters.

package-private of public classes

See description in Global Test Parameters.

private

See description in Global Test Parameters.

Common

See description in Global Test Parameters.

Static Project Initialization

See description in Global Test Parameters.

Test Case Execution

See description in Global Test Parameters.

Execute Automatic

See description in Global Test Parameters.

Execute User-Defined

See description in Global Test Parameters.

Stubs

See description in Global Test Parameters.

Options for Automatic Test Cases

See description in Global Test Parameters.

Use Automatic Stubs (white-box stubs)

See description in Global Test Parameters.

Use User Defined Stubs

See description in Global Test Parameters.

Options for User Defined Test Cases

See description in Global Test Parameters.

Use User Defined Stubs

See description in Global Test Parameters.

“Tested Set” Includes

See description in Global Test Parameters.

Stubs Class

Indicates what stub class to use while testing classes in this project. To enter the specific location of the appropriate stubs class, right-click this option, choose **Edit** from the shortcut menu, then enter the path to the stubs class.

For more information on stubs, see “Testing Classes That Reference External Resources” on page 93 and “Using Custom Stubs” on page 98.

Pre-filtering Suppression Categories

See description in Global Test Parameters.

Exceptions in Throws Clause

See description in Global Test Parameters.

DirectIllegalArgumentExceptions

See description in Global Test Parameters.

Explicitly Thrown Exceptions

See description in Global Test Parameters.

Exceptions Caught By Empty Catch

See description in Global Test Parameters.

DirectNullPointerExceptions

See description in Global Test Parameters.

Test Case Evaluation

See description in Global Test Parameters.

Report Uncaught Runtime Exceptions

See description in Global Test Parameters.

Perform Automatic Regression Testing

See description in Global Test Parameters.

Specification and Regression Test Cases

The test cases for each class should be accessed through the **Classes in Project** branch of this tree.

Project Test Parameters - Common Parameters, Search Parameters, Classes in Project

Common Parameters

See description in Global Test Parameters.

Directories

See description in Global Test Parameters.

Working directory

See description in Global Test Parameters.

Results

See description in Global Test Parameters.

Class Test Parameters Root

When you test a project, the Project Testing UI automatically creates the class test parameters for the individual classes found. This parameter determines what directory the class test parameter (.ctp) files are stored in.

The string $DEFAULT receives special treatment; it is replaced by a path relative to the location of the project test parameters (.ptp) file.

This parameter appears in most parameter levels.

javac/javac-like Parameters

See description in Global Test Parameters.

-classpath

See description in Global Test Parameters.

-cp

See description in Global Test Parameters.

System Properties

See description in Global Test Parameters.

-Xbootclasspath

See description in Global Test Parameters.

If Class is a Test Class

See description in Global Test Parameters.

Run the tests defined in the Class

See description in Global Test Parameters.

Test the Test Class itself

See description in Global Test Parameters.

Search Parameters

Contains parameters that control how Jtest searches for classes.

Skip classes already tested

If selected, Jtest will not retest a class if results for that class already exist and the class didn't change since the previous results were calculated. Jtest determines whether or not a class has changed by checking that both the .class file and the .java file contents have not changed. Timestamps are not considered.

Skip List

Opens a dialog box which lets you enter the names of specific classes that you do not want tested.

Test Only List

Opens a dialog box which lets you enter the names of the specific classes that you want tested.

Static Analysis

Contains parameters that control how Jtest searches for classes for static analysis.

Skip if .java file not found

If selected, Jtest will only perform static analysis on classes for which it finds a .java file.

Dynamic Analysis

Contains parameters that control how Jtest searches for classes for dynamic analysis.

Test public classes only

If selected, Jtest will perform dynamic analysis only on public classes.

Note that the non-public classes will be tested indirectly when called from the public classes.

Timeout

Specifies the maximum amount of time that Jtest will spend testing any one class in the project.

Classes in Project

Contains a list of all classes in the project. Also allows you to suspend and resume the finder's search for classes, and delete all individual class test parameters.

- To suspend the finder from searching for all classes in the project, right-click this node and choose **Suspend Finder** from the shortcut menu.
- To prompt the finder to resume finding classes in this project, right-click this node and choose **Resume Finder** from the shortcut menu.

- To delete all individual Class Test Parameters, right-click this node and choose **Delete All Individual Class Test Parameters** from the shortcut menu.

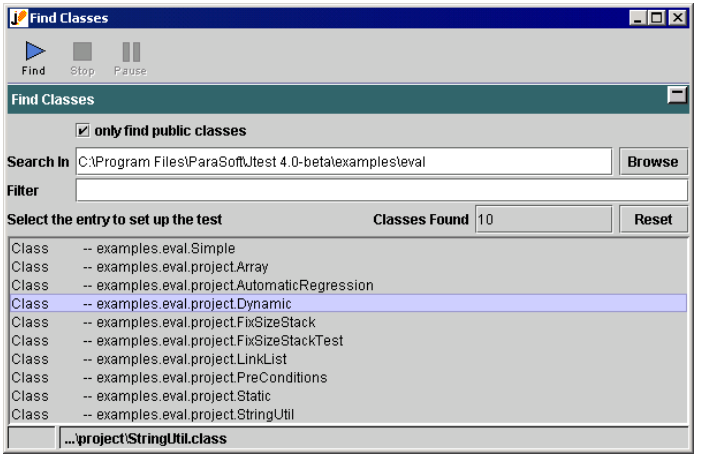
[Class Name]

Allows you to edit and reset the named class's Class Test Parameters, and open the named class in the Class Testing UI.

- To edit class test parameters, right-click this node and choose **Edit Class Test Parameters** from the shortcut menu.
- To reset all class test parameters to their default value, right-click this node and choose **Reset Class Test Parameters** from the shortcut menu.
- To load this class in the Class Testing UI (where you can focus on results for this class), right-click this node and choose **Load Test in Class Testing UI** from the shortcut menu.

Find Classes UI

The Find Classes UI searches for classes that can be tested by Jtest, then allows you to easily set up a test for any found class. This UI can be opened in the Class Testing UI by choosing **Tools> Find Classes UI**. This UI cannot be accessed from the Project Testing UI.



To find classes, tell Jtest where to start looking for classes (using the **Browse** button, or by entering the path in the **Search In** field), then click the **Start** button.

The Find Classes UI has three main components:

- The tool bar.
- The Find Classes panel.
- The status bar.

Find Classes Tool Bar

The following commands are available in the Find Class UI tool bar:

Button	Name	Action
	**Find**	Starts finding classes. The search starts in the directory, jar, or zip file specified in the **Search In** parameter.
	**Stop**	Stops finding classes.
	**Pause**	Temporarily stops finding classes. Also resumes searching after searching has been paused.

Find Classes Panel

- **Only find public classes:** If checked, Jtest will only search for public classes.

- **Search In:** Specifies where Jtest should start searching for classes to test. The parameter can be a directory, a .class file, a jar file, or a zip file.

  If the parameter is a directory, Jtest will recursively traverse the path's subdirectories, zip files, and jar files when it searches for file to test.

If the parameter is a jar or zip file, Jtest will open the file and search it for classes in which to find errors.

To browse to the directory, jar file, or zip file that you want Jtest to start searching, click the **Browse** button, locate and select the desired directory, jar file, or zip file in the file viewer, then click **Open**.

- **Filter:** Tells Jtest to find only classes that match the given expression. Use the * (asterisk) character to match zero or more characters.

  For example, if you want Jtest to look only for classes in the DB package, enter the following parameter in this field:

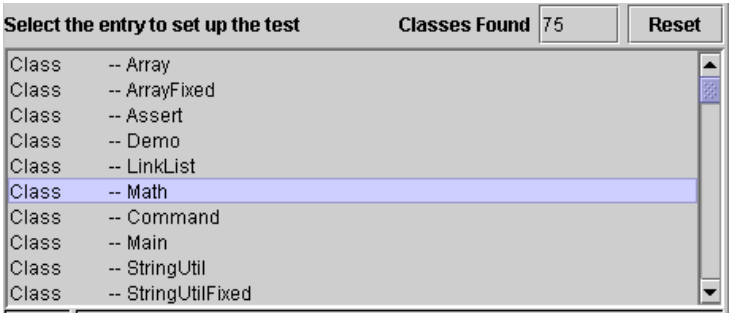
  ```
  DB.*
  ```

  When this field is left empty, Jtest will look for all classes.

- **Classes Found:** The number of classes found.

- **Reset:** Clears the lower panel.

The lower panel lists the classes that Jtest located.



To load a class found into the Class Testing UI (for testing), double-click the name of the class. The class will then be loaded in the Class Testing UI, and can be tested by clicking the **Start** button in the Class Testing UI.

Status Bar

The status bar displays the current search path.

Jtest Tutorials

Jtest's tutorials offer step-by-step guides on such topics as:

- Performing static analysis
- Performing white-box testing
- Performing black-box testing (includes automatic black-box testing and adding user-defined test cases with Test Classes and method inputs)
- Performing regression testing
- Testing a set of classes
- Using JUnit Test Classes with Jtest

These tutorials are available online at
http://www.parasoft.com/products/jtest/manuals/tutorials/html/index.htm

You can reach this page by choosing **Help> Tutorial** from either Jtest UI.

You can also access a RuleWizard tutorial at
http://www.parasoft.com/products/jtest/manuals/v4_0/rulewizard/demo.htm

Jtest FAQs

Jtest FAQs are available online at http://www.parasoft.com/products/jtest/papers/faq.htm.

Fixing Errors Found

This topic explains how and why to repair the various types of errors that Jtest finds in your code.

Learning More About Errors Found

Before you begin to fix the errors found, you should explore them to determine what caused each error. For example, if Jtest reports an uncaught runtime exception, you should examine the information related to that exception to determine whether it is the result of an incorrectly behaving method, an unexpected argument, a correctly behaving method, or a developer-use only method. If Jtest reports a coding standard violation, you should look at the information related to that violation to determine whether or not it is the result of coding standard that you want to enforce for the current project.

If you performed your test in the Project Testing UI, errors found are displayed in the Project Testing UI's Results panel. To learn more about this panel's branches and available options, see “Understanding the Results Panel” on page 45 and “Exploring and Customizing Project Test Results” on page 53.

If you performed your test in the Class Testing UI, errors found are displayed in the Class Testing UI’s Errors Found panel. To learn more about this panel's branches and available options, see “Understanding the Errors Found Panel” on page 32 and “Exploring and Customizing Class Test Results” on page 37.

Both panels classify errors found into four categories:

- Static Analysis Violations
- Design by Contract Violations
- Uncaught Runtime Exceptions
- Specification/Regression Errors

Static Analysis Violations

During static analysis, Jtest automatically tests your code for possible coding standard violations. Coding standard violations are reported under the **Static Analysis** heading. Jtest reports the following information for each violation found:

1. **Rule violation:** Jtest presents each rule violation by listing the rule, a rule ID, and a number that indicates the severity level of the rule. The rule can be one of 174 built in rules provided by Jtest or a user-defined rule.
2. **Suggestion:** To see a suggestion of how to avoid this coding standard violation, expand the rule violation branch.
3. **Line Reference:** To view a line reference of where the coding standard violation occurred, expand the suggestion branch. If you double click the line reference, the source viewer will open with the line of code that produced the violation highlighted in yellow.

Design by Contract Violations

During dynamic analysis, Jtest creates test cases that verify the specifications included in each class's DbC-format contact. Violations found are reported under the **Design by Contract Violations** heading. Design by Contract violations are organized according to the nature of the violation. This heading contains the following violation categories:

- @**pre violations:** Contains information about violations that occur when a method is called incorrectly.
- @**post violations:** Contains information about violations that occur when a method does not return the expected value.
- @**invariant violations:** Contains information about violations that occur when an @invariant contact condition is not met.
- @**assert violations:** Contains information about violations that occur when an @assert contact condition is not met.

Each error message includes file/line information as well as stack trace and calling sequence information.

Uncaught Runtime Exceptions

During dynamic analysis, Jtest automatically creates and executes test cases for each class's methods; it also executes any user-defined test cases that you have added. Exceptions found from automatic and user-defined test cases are reported under the **Uncaught Runtime Exceptions** heading. Jtest reports the following information for each exception found:

1. **Exception Description:** Jtest presents each uncaught runtime exception by listing the method that produced the exception, followed by a description of the exception that was thrown.
2. **Stack Trace:** To see the stack trace, as well as a line reference, expand the branch that displays the exception description. If you double click the line reference, the source viewer will open with the line of code that produced the exception highlighted in yellow. To see the calling sequence that produced the exception, expand the **Test Case Input** branch.
3. **Test Case:** To view the test case that Jtest generated to find this uncaught runtime exception, right click **Test Case Input** and choose **View Example Test Case** from the shortcut menu.

Specification/Regression Errors

Jtest performs automatic regression testing on a class after the class has been tested at least once. Jtest will compare results from old tests with the current test to ensure that new errors are not introduced into the code after modification. Inconsistencies between old test runs and the current test are reported under Specification and Regression Errors in the Errors Found panel of the Class Testing UI.

Fixing Errors Found

This section offers suggestions on how to fix the errors that Jtest found.

Coding Standard Enforcement (Static Analysis) Violations

Violation Needs to Be FIxed

- **Description:** The rule violation indicates a violation that needs to be fixed.
- **Repair:** Repair the code to follow the coding standard. For a complete explanation of a particular coding standard, see “Built-in Static Analysis Rules” on page 277.
- **Benefit of Repair:** Reduced opportunity for errors to enter into the code.

Violation Does Not Apply

- **Description:** The coding standard does not apply to your current project, or your development team has decided not to enforce the coding standard.
- **Repair:** Right click the node that lists the violation, then select **Disable This Rule** from the shortcut menu. The rule can be re-enabled in the Global Test Parameters.
- **Benefit of Repair:** Violations of this rule will not be reported in future tests.

Design By Contract Violations

- **Description:** Design by Contract violations indicate either an error in the code or an error in the contract.
- **Repair:** Determine if the problem is in the code or in the contract, then make the appropriate modifications.
- **Benefit of Repair:** The code or contract has been fixed.

White-Box Testing Errors

Incorrectly Behaving Method

- **Description:** The method is behaving incorrectly; the method shouldn't throw an exception for those arguments.
- **Repair:** Repair the method's code so that it behaves correctly.
- **Benefit of Repair** The code has been fixed.

Unexpected Arguments

- **Description:** The method is not supposed to handle those arguments; an exception is thrown because the method is not expecting those arguments.
- **Repair:** If the arguments are illegal, either add an @pre condition to the method (this is the recommended repair), or throw an IllegalArgumentException.
- **Benefit of Repair:** The code is documented, easier to maintain, and easier to change. The code explicitly says what arguments the method handles and which ones it doesn't. The error messages when using illegal arguments are clarified. Encapsulation is enforced. When the method is passed arguments that it is not supposed to handle, it should throw an IllegalArgumentException. If this is not done, one of the following things will occur:
    - The method throws an exception exposing internal implementation details of the method, hence, violating encapsulation.

- The method doesn't throw an exception when passed illegal arguments. Instead it returns a value without any meaning or leaves the program in an inconsistent state.

Correctly Behaving Methods

- **Description**: The method is behaving correctly; the output of the method is to throw an Exception.
- **Repair:** Either add an @exception tag to the document that the method throws that exception (this is the recommended repair), or specify the type of exception in the method's throws clause.
- **Benefit of Repair:** The code is documented (the code explicitly says that the method can throw that kind of exception) and easier to maintain. Someone looking at the code later on will know whether the method is throwing an exception because the code has a bug or because the code is supposed to throw an exception.

Developer Use Only Methods

- **Description:** The method is never sent those arguments.
- **Repair:** Do one of the following:
  - Add an @pre condition to that method (this is the recommended repair).
  - Decrease the accessibility of the method (e.g., set it to private).
  - Throw an IllegalArgumentException.
- **Benefit of Repair:** Someone looking at the code later on will know whether the program is throwing an exception because the code is incorrect or because the code is not supposed to handle those arguments.

Specification/Regression Errors

Modification-Related Error

- **Description:** Jtest reports a specification/regression error for code that you recently modified.
- **Repair:** If the result is unexpected, fix the code to restore the original functionality. If it is expected, change the reference value that Jtest uses for the related test case.
- **Benefit of Repair:** Errors are removed or Jtest checks the test case against the correct reference value in future test runs.

Built-in Static Analysis Rules

Jtest includes the following built-in rules. A detailed description of each rule is provided in the pages that follow. Rules are listed in alphabetical order.

Coding Standard Rules

Design by Contract Rules

EJB Rules

Garbage Collection Rules

Global Static Analysis Rules

Initialization Rules

Internationalization Rules

Javadoc Comment Rules

Class Metrics

Miscellaneous Rules

Naming Convention Rules

Object Oriented Programming Rules

Optimization Rules

Possible Bugs Rules

Project Metrics

Portability Rules

Security Rules

Servlet Rules

Threads and Synchronization Rules

Unused Code Rules

CODSTA.CLS

Place constants on the left side of comparisons

Description

This rule flags code that does not place constants on the left side of comparisons.

Example

```
package CODSTA;

public class CLS {
    public void testMethod (int something) {
        if (something == 5) {}  // violation.
    }
}
```

Repair

Place constants in left side of comparisons.

```
    public void testMethod (int something) {
        if (5 == something ) {}
    }
```

Reference

Section 2.5.2 of http://www.AmbySoft.com/javaCodingStandards.pdf

CODSTA.CRS

Place constants on the right side of comparisons

Description

This rule flags code that does not place constants on the right side of comparisons.

Example

```
package CODSTA;

public class CRS {
    public void testMethod (int something) {
        if (5 == something) {}  // violation.
    }
}
```

Repair

Place constants in right side of comparisons.

```
    public void testMethod (int something) {
        if (something == 5) {}
    }
```

CODSTA.DCI

Define constants in an “interface”

Description

This rule flags any non-private named constant that is not defined in an “interface”.

Example

```
package CODSTA;

class DCI{
    private int[] getArray () {
        return new int [ARRAY_SIZE];
    }
      static final int ARRAY_SIZE = 1000;
}
```

Repair

```
class USN_fixed {
    private int[] getArray () {
        return new int [Constants.ARRAY_SIZE];
    }
}
interface Constants {
    int ARRAY_SIZE = 1000;
}
```

CODSTA.DCTOR

Define a default constructor whenever possible

Description

This rule flags code that define a default constructor, but does not.

Default constructors allow classes of unknown types to be dynamically loaded and instantiated at compile time (as is done when loading unknown Applets from html pages). In Java 1.1 and higher, reflection somewhat alleviates the need for no-argument constructors, but many classes that dynamically instantiate other classes at runtime still depend on their presence.

Example

```
package CODSTA;

public class DCTOR {  // missing a default constructor.
    public DCTOR(int size) { _size = size; }
    private int _size;
}
```

Repair

Define default constructor (no argument).

Reference

http://www.infospheres.caltech.edu/resources/code_standards/recommendations.html

CODSTA.IMPT

Minimize * form of “import” statements

Description

This rule flags code that uses * in import statements.

We recommend that you try to “import” only necessary classes and be precise about what you are importing. Otherwise, readers might have a difficult time understanding its context and dependencies. Some people even prefer not using “import” at all (thus requiring that every class reference is fully qualified); this prevents all possible ambiguity and reduces source code changes if package names change.

Example

```
package CODSTA;
import java.io.*;

public class IMPT {
    void method (InputStream in) {
        if (in == null) return;
    }
}
```

Reference

http://g.oswego.edu/dl/html/javaCodingStd.html

CODSTA.IMPT2

Use wild card symbols when importing classes

Description

This rule flags code that does not use wild card symbols when importing classes.

The import statement allows the use of the wild cards when indicating the names of classes. For example, the statement import java.awt.*; brings in all of the classes in the package java.awt at once. Actually, that's not completely true. What really happens is that every class that you use form the java.awt package will be brought into your code when it is compiled, classes that you do not use will not be.

Note: Jtest has another rule, CODSTA.IMPT, which discourages the use of wild card symbols in import statements. Because we have two references, which contain conflicting opinions, ParaSoft added both rules for the users to decide.

Example

```
package CODSTA;
import java.io.InputStream;  // violation.

public class IMPT2 {
    void method (InputStream in) {
        if (in == null) return;
    }
}
```

Reference

Section 7.2 of http://www.AmbySoft.com/javaCodingStandards.pdf

CODSTA.ISACF

Avoid using an "interface" to define constants

Description

This rule flags code where an “interface” is used to define a constant.

An interface should not be used to define constants. While it is common practice to use a constant interface (an interface that only contains static final fields and no methods) this should be avoided. The use of constants is an implementation detail. Implementing a constant interface causes this implementation detail to leak into the class's exported API. In order to avoid this, constants should be moved to a utility class, a class which only contains static variables and methods.

Note: Jtest’s CODSTA.DCI rule conflicts with this rule. Because we have seen arguments for both guidelines, we have included both rules and will allow you to decide which one to use.

Example

```
package CODSTA;

public interface ISACF {
    int NUM = 1234; // violation
}
```

Repair

Place constants in a utility class instead of an "interface"

```
package CODSTA;

public class ISACF_CLASS {
    private ISACF_CLASS() {}  // Prevents instantiation
```

```
    static final int NUM = 1234;
}
```

Reference

Bloch, Joshua. *Effective Java Programming Language Guide.* Addison Wesley, 2001, pp 89 - 90.

CODSTA.LONG

Use ‘L’ instead of ‘l’ to express “long” integer constants

Description

This rule flags code where a long integer constant is indicated by ‘l’ instead of by ‘L’.

Integer constants are “long” if they end in ‘L’ or ‘l’;. ‘L’ is preferred over ‘l’ because ‘l’ (lowercase ‘L’) can easily be confused with ‘1’ (the number one).

Example

```
package CODSTA;

class LONG {
    long getLongNumber () {
        long temp = 23434l; // Is this 23434L or 234341?
        return temp;
    }
}
```

Repair

```
class LONG {
    long getLongNumber () {
        long temp = 23434L;
        return temp;
    }
}
```

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp.108

CODSTA.MAIN

The ’main()’ method must be “public”, “static” and “void”

Description

This rule flags any ‘main()’ method that is not “public”, “static”, and “void”.

The ‘main()’ method must be “public”, “static”, and “void”, and it must accept a single argument of type String[].

Example

```
package CODSTA;

public class MAIN {
   static void main(String[] args) {// main method is
                                          //not a public.
        System.out.println("hello");
    }
}
```

Repair

Use following signature for a ‘main()’ method.

```
public static void main(String[] args)
```

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp.55-56.

CODSTA.MVOS

Do not declare multiple variables in one statement

Description

This rule flags code where multiple variables are declared in one statement. Declaring too many variables in one statement can make code confusing.

Example

```
package CODSTA;

class MVOS {
   public void foo() {
      int aaa, ccc;  //violation
   }
}
```

Repair

```
package CODSTA;
class MVOS {
   public void foo() {
      int aaa;
      int ccc;
   }
}
```

CODSTA.NCAC

Never call an “abstract” method from a constructor in an “abstract” class

Description

This rule flags code that calls an “abstract” method from a constructor in an “abstract” class.

Calling abstract methods from an "abstract" class's constructor causes the object's methods to be used before it is finished using its constructors.

Example

```
package CODSTA;

abstract class NCAC {
    public NCAC () {
        System.out.println("Constructor: ");
        test (); // invoke abstract method from the constructor.
    }
    abstract public void test ();
}
class MyClass extends NCAC {
    public MyClass (int size) {
        super ();
        System.out.println("setting size to : " +size);
        _size = size;
    }
    public void test () {
        _size++;
        System.out.println("Increament : " +_size);
    }
    private int _size = 0;
}

Result of the above code is:
 MyClass mc = new MyClass (50);
```

```
Constructor:
Increment : 1        // super class's constructor called test();
setting size to : 50  // finish executing MyClass' constructor.
```

Reference

Warren, Nigel, and Bishop, Philip. *Java in Practice*. Addison-Wesley, 1999, pp.103-104.

CODSTA.NCE

Never use ‘Exception’, RuntimeException’, or ‘Throwable’ in “catch” statement

Description

This rule flags code that tries to “catch” an ‘Exception’, ‘RuntimeException’, or ‘Throwable’.

‘Exception’, ‘RuntimeException’, and ‘Throwable’ are too general.

Example

```
package CODSTA;

public class NCE {
    void method () {
        try {
        } catch (Exception e1) {
        }
        try {
        } catch (RuntimeException e2) {
        }
        try {
        } catch (RuntimeException e3) {
        }
    }
}
```

Repair

Deal with subclasses of ‘Exception’, ‘RuntimeException’, and ‘Throwable’ when handling exceptions.

Reference

Warren, Nigel, and Bishop, Philip. *Java in Practice*. Addison-Wesley, 1999, pp. 68-69.

CODSTA.NTE

Never “throw” an ‘Error’ directly

Description

This rule flags code that “throw”s an ‘Error’ directly.

“Throwing” an ‘Error’ directly is too general.

Example

```
package CODSTA;

public class NTE {
    void method () {
        throw new LinkageError ();
    }

    void method2 () {
        throw new Error ();
    }
}
```

Repair

Explicitly deal with the subclasses of a superclass derived from ‘Error’ to avoid potential bugs.

CODSTA.NTX

Never “throw” class ‘Exception’ directly

Description

This rule flags code where ‘Exception’ is being used too generally.

Example

```
package CODSTA;

public class NTX {
    void method () throws Exception, ArithmeticException {
        throw new Exception ();
    }

    void foo () throws Exception {
        Exception e = new Exception ("TEST");
        throw e;
    }
}
```

Repair

Concentrate on handling subclasses of ‘Exception.’

Reference

Warren, Nigel, and Bishop, Philip. *Java in Practice*. Addison-Wesley, 1999, pp. 76.

CODSTA.OGM

Organize member variables by name

Description

This rule flags code where group members with the same name are not ordered together.

This rule enforces various standards to improve readability.

Example

```
package CODSTA;

public class OGM {
    void foo () {}
    void bar () {}
    void foo (int a) {
    }
}
```

Repair

Move the members called “foo()” together.

CODSTA.OVERRIDE

If you override 'Object.equals()', you should also override 'Object.hashCode()'

Description

This rule flags code where a class that overrides 'Object.equals()' does not also override 'Object.hashCode()'.

A class that overrides 'Object.equals()' should also override 'Object.hashCode()'. Containers and other utilities that group or compare objects in ways depending on equality rely on hashcodes to indicate possible equality.

Example

```
package CODSTA;

public class OVERRIDE { // violation, no hashCode()
    public Object equals (Object o) {
        // implementations
    }
}
```

Repair

Whenever a class overrides an equals() method, it should also override hashCode().

Reference

http://www.infospheres.caltech.edu/resources/code_standards/recommendations.html

CODSTA.PML

Put the ‘main()’ method last

Description

This rule flags code where the ‘main()’ method is not last item in the class definition.

This rule makes the program comply with various coding standards regarding the form of class definitions.

Example

```
package CODSTA;

class PML {
    public static void main (String args[]) {
    }
    void foo () {
    }
}
```

Repair

List method ‘main()’ last within the “class” definition.

CODSTA.SMC

Avoid “switch” statements with many “cases”

Description

This rule flags any “switch” statement with many “cases”.

“switch” statements with many “case” statements make code difficult to follow. More importantly, switches with many cases often indicate places where polymorphic behavior could better be used to provide different behavior for different types. Note that although the general principle is to avoid many cases in a switch, the actual cutoff point is arbitrary.

Example

```
package CODSTA;

class SMC
{
   public void foo(int i) {
      switch (i) {          // Violation
      case 1:
         break;
      case 2:
         break;
      case 3:
         break;
      case 4:
         break;
      case 5:
         break;
      case 6:
         break;
      case 7:
         break;
      case 8:
         break;
```

```
      case 9:
         break;
      case 10:
         break;
      case 11:
         break;
      default:
         break;
      }
   }
}
```

Repair

Look for cleaner ways to invoke the alternative behaviors.

CODSTA.UCC

Utility classes should only have "private" constructors

Description

This rule flags any utility class that has non-”private” constructors.

A utility class only contains static methods and static variables. Because the utility class is not designed to be instantiated, all of the constructors should be private.

Example

```
package CODSTA;

public class UCC {

    public static String s = "UCC";

    public UCC() {} // violation

    public static String getUCC() {
        return "UCC";
    }
}
```

Repair

```
package CODSTA;

public class UCC {

    public static String s = "UCC";
    private UCC() {}

    public static String getUCC() {
```

```
        return "UCC";
    }
}
```

Reference

Bloch, Joshua. *Effective Java Programming Language Guide.* Addison Wesley, 2001, pp 89 - 90.

CODSTA.UCDC

Utility classes should not have a default constructor

Description

This rule flags any utility class that has a default constructors.

A utility class only contains static methods and static variables. Because an implicit default constructor is "public" and a utility class is not designed to be instantiated, the "public" default constructor of a utility class should be declared "private".

Example

```
package CODSTA;

public class UCDC {

    // violation: implicit default constructor is public

    public static String UCDC = "UCDC";

    public static String getUCDC() {
        return "UCDC";
    }
}
```

Repair

```
package CODSTA;

public class UCDC {

    private UCDC() {}

    public static String UCDC = "UCDC";
```

```
    public static String getUCDC() {
        return "UCDC";
    }
}
```

Reference

Bloch, Joshua. *Effective Java Programming Language Guide.* Addison Wesley, 2001, p. 12

CODSTA.USN

Use symbolic names for constants

Description

This rule flags code that uses an unnamed constant.

Named constants (final static variables) make the code much easier to understand and maintain. To avoid reporting too many spurious errors, Jtest will not report an error for the following integer constants:

-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Jtest will only give one error per constant found in the code.

Example

```
package CODSTA;

class USN {
    private int[] getArray () {
        return new int [1000];
    }
}
```

Repair

```
class USN {
    private int[] getArray () {
        return new int [ARRAY_SIZE];
    }
    private static final int ARRAY_SIZE = 1000;
}
```

CODSTA.VDT

Do not declare multiple variables of different types in one statement

Description

This rule flags any code where multiple variables of different types are declared in a single declaration statement.

Declaring multiple variables of different types in a single declaration statement can cause confusion.

Example

```
package CODSTA;
class VDT
{
   public void foo() {
      int aaa, bbb[];        // Violation
      int ccc, ddd;
   }
}
```

Repair

```
package CODSTA;

class VDT
{
   public void foo() {
      int aaa;
      int bbb[];
      int ccc, ddd;
   }
}
```

DBC.PKGC

All package-private classes should have the '@invariant' contract

Description

This rule flags any package-private class without an '@invariant' contract: '$name'.

Example

```
package DBC;

class PKGC {
   /**
     * @pre size >= 0
     * @post ($return != null && $pre (size + 1) == size + 1)
     * @exception NegativeArraySizeException size < 0
     */
    public int[] allocate (int size) {
        return new int [size];
    }
    public String toString () {
        return "PKGC";
    }
    public static void main (String[] args) {
        new Strategy ().allocate (5);
    }
}
```

Repair

Provide the ‘@invariant’ Javadoc tag.

```
/** @invariant toString ().equals ("PKGC") */
```

DBC.PKGMPOST

All package-private methods should have the '@post' contract

Description

This rule flags any package-private method that does not have an ‘@post’ contract in its Javadoc.

Example

```
/** @invariant toString ().equals ("Strategy") */
class PKGMPOST {
   /**
     * @pre size >= 0
     * @exception NegativeArraySizeException size < 0
     */
    int[] allocate (int size) {
        return new int [size];
    }
}
```

Repair

Provide the ‘@post’ Javadoc tag.

```
/**
  * @pre size >= 0
  * @post ($return != null && $pre (size +1) == size + 1)
  * @exception NegativeArraySizeException size < 0
  */
```

DBC.PKGMPRE

All package-private methods should have the '@pre' contract

Description

This rule flags any package-private method that does not have an ‘@pre’ contract in its Javadoc.

Example

```
package DBC;

/** @invariant toString ().equals ("Strategy") */
class PKGMPRE {
   /**
     * @post ($return != null && $pre (size + 1) == size + 1)
     * @exception NegativeArraySizeException size < 0
     */
    int[] allocate (int size) {
        return new int [size];
    }
}
```

Repair

Provide the ‘@pre’ condition.

```
   /**
     * @pre size >= 0
     * @post ($return != null && $pre (size + 1) == size + 1)
     * @exception NegativeArraySizeException size < 0
     */
```

DBC.PPIC

All "private" classes should have the '@invariant' contract

Description

This rule flags any “private” class that does not have an ‘@invariant’ contract.

Example

```
package DBC;

public class OrderedList {
    /**
      * @pre["item is not in list"] contains (item) == false
      * @post["item is in list"] contains (item) == true
      */
    public void insert (PRIC item) {
        // ...
    }
    public boolean contains (PRIC item) {
        //NYI:
        return false;
    }
}

private class PRIC {
    PRIC (int value) {
        _value = value;
    }
    int getValue () {
        return _value;
    }
    void setNext (PRIC next) {
        _next = next;
    }
    PRIC getNext () {
```

```
        return _next;
    }
    private int _value;
    private PRIC _next;
}
```

Repair

Provide the ‘@invariant’ contract.

DBC.PRIMPOST

All "private" methods should have the '@post' contract

Description

This rule flags any “private” method that does not have an ‘@post’ contract in its Javadoc.

Example

```
package DBC;
public class PRIMPOST {
    /**
      * @pre size () < MAX_SIZE - 1
      * @post peek () == object
      * @post size () == $pre (int, size ()) + 1
      * @concurrency sequential
      */
    public void push (Object object) {
        _storage [_top++] = object;
    }
    /**
      * @pre size () > 0
      */
    public Object peek () {
        return _storage [_top - 1];
    }
    public int size () {
        return _top;
    }
    private boolean isEmpty () {
        return _top == 0;
    }

    private Object _storage[] = new Object [MAX_SIZE];
    private int _top;
    private final static int MAX_SIZE = 100;
```

```
}
```

Repair

Provide the ‘@post’ contract.

```
/** @post $return == (size () == 0) */
```

DBC.PRIMPRE

All "private" methods should have the '@pre' contract

Description

This rule flags any “private” method that does not have an ‘@pre’ contract in its Javadoc.

Example

```
package DBC;
public class PRIMPRE {
    /**
      * @pre size () < MAX_SIZE - 1
      * @post peek () == object
      * @post size () == $pre (int, size ()) + 1
      * @concurrency sequential
      */
    public void push (Object object) {
        _storage [_top++] = object;
    }
    /** @pre size () > 0 */
    public Object peek () {
        return _storage [_top - 1];
    }
    public int size () {
        return _top;
    }
    /** @post $return == (size () == 0) */
    private boolean isEmpty () {
        return _top == 0;
    }

    private Object _storage[] = new Object [MAX_SIZE];
    private int _top;
    private final static int MAX_SIZE = 100;
}
```

Repair

Provide the ‘@pre’ contract.

DBC.PROC

All "protected" classes should have the '@invariant' contract

Description

This rule flags any "protected" class that does not have an '@invariant' contract.

Example

```
//From: "The Pragmatic Programmer", p.110.
/**
  * @Date 2000/
  */
package DBC;

protected class PROC {
    /**
      * @pre["item is not in list"] contains (item) == false
      * @post["item is in list"] contains (item) == true
      */
    public void insert (Item item) {
        // ...
    }
    public boolean contains (Item item) {
        //NYI:
        return false;
    }
}

class Item {
    Item (int value) {
        _value = value;
    }
    int getValue () {
        return _value;
    }
```

```
    void setNext (Item next) {
        _next = next;
    }
    Item getNext () {
        return _next;
    }
    private int _value;
    private Item _next;
}
```

Repair

Provide the ‘@invariant’ contract.

```
/**
  * @invariant["items are ordered"] {
  *    for (Enumeration e = elements; e.hasMoreElements (); ) {
  *        Item item = (Item) e.nextElement ();
  *        if (item.getNext () != null)
  *            $assert (item.getValue () < item.getNext ().getValue
()));
  *    }
  * }
  */
```

DBC.PROMPOST

All "protected" methods should have the '@post' contract

Description

This rule flags any "protected" class that does not have an ‘@post' contract in its Javadoc.

Example

```
package DBC;
protected class PROMPOST {
    /**
         @pre length > 0
      */
    protected void method (int length) {
        array = new [length];
        // do something.
    }
    private int[] array;
}
```

Repair

Provide the ‘@post’ contract in “protected” methods.

DBC.PROMPRE

All "protected" methods should have the '@pre' contract

Description

This rule flags any "protected" method that does not have an ‘@pre’ contract in its Javadoc.

Example

```
package DBC;
public class PROMPRE {
    /** @param   none
        @post    none
      */
    protected void method () {
        // do something.
    }
}
```

DBC.PUBC

All "public" classes should have the '@invariant' contract

Description

This rule flags any "public" class that does not have an ‘@invariant’ contract.

Example

```
//From: "The Pragmatic Programmer", p.110.
/**
  * @author
  */

public class PUBC {
    /**
      * @pre["item is not in list"] contains (item) == false
      * @post["item is in list"] contains (item) == true
      */
    public void insert (Item item) {
        // ...
    }
    public boolean contains (Item item) {
        //NYI:
        return false;
    }
}

class Item {
    Item (int value) {
        _value = value;
    }
    int getValue () {
        return _value;
    }
    void setNext (Item next) {
```

```
        _next = next;
    }
    Item getNext () {
        return _next;
    }
    private int _value;
    private Item _next;
}
```

Repair

Provide the '@invariant' Javadoc tag.

```
/**
  * @invariant["items are ordered"] {
  *    for (Enumeration e = elements; e.hasMoreElements (); ) {
  *        Item item = (Item) e.nextElement ();
  *        if (item.getNext () != null)
  *            $assert (item.getValue () < item.getNext ().getValue
());
  *    }
  * }
  */
```

DBC.PUBMPOST

All "public" methods should have the '@post' contract

Description

This rule flags any "public" class that does not have an ‘@post contract in its Javadoc.

Example

```
package DBC;
public class PUBMPOST {
    /**
      * @pre size () < MAX_SIZE - 1
      * @post peek () == object
      * @post size () == $pre (int, size ()) + 1
      * @concurrency sequential
      */
    public void push (Object object) {
        _storage [_top++] = object;
    }
    /**
      * @pre size () > 0
      */
    public Object peek () {
        return _storage [_top - 1];
    }
    public int size () {
        return _top;
    }

    private Object _storage[] = new Object [MAX_SIZE];
    private int _top;
    private final static int MAX_SIZE = 100;
}
```

Repair

Provide the ‘@post’ contract.

```
/**
  * @pre size () > 0
  * @post size () == $pre (int, size ())
  */
```

DBC.PUBMPRE

All "public" methods should have the '@pre' contract

Description

This rule flags any "public" method that does not have an ‘@pre’ contract in its Javadoc.

Example

```
package DBC;
public class PUBMPRE {
    /**
      * @post peek () == object
      * @post size () == $pre (int, size ()) + 1
      * @concurrency sequential
      */
    public void push (Object object) {
        _storage [_top++] = object;
    }
    /**
      * @pre size () > 0
      */
    public Object peek () {
        return _storage [_top - 1];
    }
    public int size () {
        return _top;
    }

    private Object _storage[] = new Object [MAX_SIZE];
    private int _top;
    private final static int MAX_SIZE = 100;
}
```

Repair

Provide the ‘@pre’ contract.

```
/**
  * @pre size () < MAX_SIZE - 1
  * @post peek () == object
  * @post size () == $pre (int, size ()) + 1
  * @concurrency sequential
  */
```

EJB.AMSC

Avoid accessing or modifying security configuration objects

Description

This rule flags Bean classes that access or modify security configuration objects. By following this rule, you can prevent security problems.

Reference

Sanjay Mahapatra, “Programming restrictions on EJB.” *JavaWorld*, August 2000.

EJB.CDP

A Bean "class" should be declared as "public"

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Session.fm.html

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Entity.fm.html

EJB.CNDA

A Bean "class" cannot be declared as "abstract"

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Session.fm.html

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Entity.fm.html

EJB.CNDF

Bean "class" cannot be declared as "final"

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Session.fm.html

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Entity.fm.html

EJB.CRTE

ejbCreate() must be “public”, and cannot be “static” or “final”

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Session.fm.html

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Entity.fm.html

EJB.FNDM

Finder methods cannot be "final" or "static" and they must be “public”

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/tech-docs/guides/ejb/html/Entity.fm.html

EJB.IECM

Bean "class" should implement one or more ‘ejbCreate()’ methods

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Session.fm.html

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Entity.fm.html

EJB.IEPM

An EntityBean "class" should implement one or more ‘ejbPostCreate()’ methods

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/tech-docs/guides/ejb/html/Entity.fm.html

EJB.LNL

Avoid loading native libraries in Bean class

Description

This rule flags code that loads native libraries in a Bean class.

Reference

Sanjay Mahapatra, “Programming restrictions on EJB.” *JavaWorld*, August 2000.

EJB.MNDF

A Bean “class” must not define the ‘finalize()’ method

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Session.fm.html

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Entity.fm.html

EJB.NFS

Declare all "static" fields in the EJB component as "final"

Description

This rule flags any non-final “static” field in a Bean class.

Declaring all "static" fields in the EJB component as "final" ensures consistent runtime semantics so that EJB containers have the flexibility to distribute instances across multiple JVMs.

Reference

Sanjay Mahapatra, “Programming restrictions on EJB.” *JavaWorld*, August 2000.

EJB.PCRTE

‘ejbPostCreate()’ must be “public”, and it cannot be “static” or “final”

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/tech-docs/guides/ejb/html/Entity.fm.html

EJB.RT

The finder methods' "return" type must be the primary key or a collection of primary keys

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/tech-docs/guides/ejb/html/Entity.fm.html

EJB.RTC

The SessionBean's ‘ejbCreate()’ methods’ “return” type must be “void”

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Session.fm.html

EJB.RTP

The ‘ejbPostCreate()’ method’s “return” type must be “void”

Description

See the reference.

Reference

Enterprise JavaBeans Developer's Guide

http://web2.java.sun.com/j2ee/j2sdkee/tech-docs/guides/ejb/html/Entity.fm.html

EJB.RUH

Reuse EJB homes

Description

This rule flags code that does not reuse EJB homes but should. This rule only applies to simple applications.

EJB homes are obtained from the WebSphere Application Server through a JNDI naming lookup. This is an expensive operation that can be minimized by caching and reusing EJB Home objects.

Example

```
package EJB;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.rmi.*;
import javax.naming.*;

public class RUH extends HttpServlet {
    public void transaction () throws ServletException {
        Context ctx = null;
        try {
            ctx = new InitialContext (new java.util.Hashtable ());
            Object homeObject = ctx.lookup ("EJB JNDI NAME");
            //violation, Home interface should not be a local
             // variable.
            AccountHome aHome = (AccountHome)Portable
RemoteObject.narrow (
                    homeObject, AccountHome.class);
        } catch (Exception e) {
            throw new ServletException ("INIT ERROR" +e.getMessage
(), e);
        } finally {
            try {
                if (ctx != null) ctx.close ();
            }  catch (Exception e) {}
```

```
        }
    }
}
```

Repair

Cache EJB Home in Servlet init method.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;

public class rw137_correct extends HttpServlet {
    private AccountHome aHome = null;  // cache Home interface.
    public void init (ServletConfig config) throws ServletException
{
        super.init (config);
        Context ctx = null;
        try {
            ctx = new InitialContext (new java.util.Hashtable ());
            Object homeObject = ctx.lookup ("EJB JNDI NAME");
            aHome = (AccountHome)javax.rmi.PortableRemoteObject.nar-
row (
                    homeObject, AccountHome.class);
        } catch (Exception e) {
            throw new ServletException ("INIT ERROR" +e.getMessage
(), e);
        } finally {
            try {
                if (ctx != null) ctx.close ();
            } catch (Exception e) {}
        }
    }
}
```

For simple applications, it might be enough to acquire the EJB home in the `servlet init()` method.

More complicated applications might require cached EJB homes in many servlet and EJBs. In these cases, you might want to create an EJB Home Locator and Caching class.

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

EJB.THISARG

Avoid passing the "this" reference as an argument

Description

This rule flags code that passes the “this” reference as an argument.

Instead of passing the “this” reference as an argument, use getEJBObject () available in SessionContext or EntityContext.

Reference

Sanjay Mahapatra, “Programming restrictions on EJB.” *JavaWorld*, August 2000.

EJB.THISRET

Avoid returning the "this" reference as a result

Description

This rule flags code that returns the “this” reference as a result. Instead if returning the “this” reference as a result, use getEJBObject () available in SessionContext or EntityContext.

Reference

Sanjay Mahapatra, “Programming restrictions on EJB.” *JavaWorld*, August 2000.

EJB.THREAD

Avoid starting, stopping, or managing threads in any way

Description

This rule flags code that calls a thread method from an EJB class.

Starting, stopping, or managing threads usage restriction eliminates the possibility of conflicts with the EJB container's responsibility of managing locking, threading, and concurrency issues.

Reference

Sanjay Mahapatra, “Programming restrictions on EJB.” *JavaWorld*, August 2000.

GC.AUTP

Avoid unnecessary temporaries when converting primitive types to String

Description

This rule flags code where an unnecessary temporary is used when converting primitive types to String.

Java provides wrapping classes for the primitive types. Those classes provide a static method ‘toString (...)’ to convert the primitive type into their String equivalent. Use this method instead of creating an object of the wrapping class and then using the instance ‘toString ()’ method.

Example

```
package GC;

public class AUTP {
   String foobar (int x) {
      return new Integer (x).toString ();  // Violation
   }
}
```

Repair

```
class AUTP_fixed {
   String foobar (int x) {
      return Integer.toString (x);
   }
}
```

GC.DUD

Do not use ‘Date[]’; use ‘long[]’ instead

Description

This rule flag any occurrence of ‘Date[]’.

Do not use arrays of ‘Date’ objects. Using an array of “long" is more efficient than using an array of ‘Date’s .

Example

```
package GC;

import java.util.*;

public class DUD {
   Date d[];        // Violation
}
```

Repair

Use an array of “long” objects instead.

GC.FCF

Make sure that ‘finalize ()’ calls ‘super.finalize ()’

Description

This rule flags code where a ‘finalize()’ method does not call ‘super.finalize()’.

Example

```
package GC;
class FCF {
    protected void finalize () throws Throwable {
    }
}
```

Repair

Add the call to ‘super.finalize()’. If that call is not made, the finalize methods of the superclasses will not be invoked. See the reference for a detailed explanation of why ‘super.finalize ()’ should be called even if the base class doesn't define ‘finalize ()’.

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp.49.

GC.FM

’finalize()’ method should not unregister listeners

Description

This rule flags code where the ‘finalize()’ method unregisters listeners.

The 'finalize()' method only gets called when there are no more references to the object. If the listeners are removed in the 'finalize()' method, the object will not be removed during garbage collection.

Example

```
package GC;
import java.awt.Panel;
import java.awt.Button;
public class FM extends Panel {
    public void finalize() {
        remove (_button);               // violation
        super.finalize();
    }
    private Button _button = new Button();
}
```

Repair

Do not call methods that remove listeners in the ’finalize()’ method body.

GC.GCB

Reuse objects; 'getClipBounds()' should not be called too often

Description

This rule flags code that calls 'getClipBounds()' too often.

The 'getClipBounds()' method always returns a new rectangle. Allocating more memory every time this method is called overworks the garbage collector.

Note: Violations are only reported for methods with more than one 'getClipBounds()'.

Example

```
package GC;
import java.awt.Graphics;

public class GCB {
    public void paint(Graphics g) {
        int firstColLine = g.getClipBounds().x;
        int lastColLine = g.getClipBounds().x + g.getClip-
Bounds().width;
    }
}
```

Repair

```
package GC;

import java.awt.Graphics;
import java.awt.Rectangle;

public class GCB {
    public void paint(Graphics g) {
        Rectangle rec = g.getClipBounds();// instantiate a object,
```

```
          rec
        int firstColLine = rec.x; // reuse "rec"
        int lastColLine = rec.x + rec.width;   // reuse "rec"
    }
}
```

GC.IFF

Invoke ‘super.finalize()’ in the ‘finally’ blocks of ‘finalize()’ methods

Description

This rule flags code where the “finally” block of a ‘finalize()’ method does not invoke ‘super.finalize()’.

Example

```
package GC;

class IFF {
    public void finalize() throws Throwable {
       try {
       } catch (Exception e) {
       } finally {
       }                        // Violation
    }
}
```

Repair

```
class BetterIFF {
    public void finalize() throws Throwable {
       try {
       } catch (Exception e) {
       } finally {
          super.finalize();
       }
    }
}
```

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp. 49.

GC.NCF

Never call ‘finalize()’ explicitly

Description

This rule flags code that explicitly calls ‘finalize()’.

Calling the ‘finalize()’ method explicitly insures that ‘finalize()’ is called, but the Garbage Collector during runtime will call the ‘finalize()’ method again when the object is collected.

Example

```
package GC;

class NCF {
    public void finalize() throws Throwable {
        super.finalize();
    }
}
class Test {
    void closeTest () throws Throwable {
        _test.finalize(); // this may get called again by Java
                          //Virtual Machine
        _test = null;
    }
    private NCF _test = new NCF();
}
```

Repair

Create a method to handle the release of the memory, then call this method from a ‘finalize()’ method before calling the ‘super.finalize()’ method.

```
public void release() {
    if (!closed) {
        _test.finalize ();
```

```
        closed = true;
    }
}
public void finalize () throws Throwable {
    release ();
    super.finalize();
}
```

Reference

Warren, Nigel, and Bishop, Philip. *Java in Practice.* Addison-Wesley, 1999, pp. 110-111.

GC.OSTM

Be aware of memory leaks due to ‘ObjectStream’ usage

Description

This rule flags any case where ObjectStream usage might cause memory leaks.

ObjectStreams are designed to handle cases where the same Object is sent across a connection multiple times. For this reason, ObjectStream classes keep a reference to all objects written or read until the ‘reset()’ method is called. Those objects will not be garbage collected until ‘reset()’ is called.

Example

```
package GC;
import java.io.*;

public class OSTM {
    public void writeToStream(ObjectOutputStream os, String s)
        throws IOException {
        os.writeObject (s);
    }
}
```

Repair

Use ‘reset()’ of ObjectOutputStream or InputStream class’s method to clear the list of Objects written to the Stream. Or, use DataStreams instead of ObjectStreams in terms of Strings or byte arrays for optimal performance.

```
package GC;
import java.io.*;

public class OSTM {
    public void writeToStream(ObjectOutputStream os, String s)
        throws IOException {
        os.writeObject (s);
        os.reset();            // prevents memory leaks.
    }
}
```

GC.STV

Avoid “static” collection; it can grow without bounds

Description

This rule flags any occurrence of “static” collection.

Static variables that can hold large number of objects (e.g., static variables of type Vector or Hashtable) are prime candidates for memory leaks.

Example

```
package GC;
import java.util.Vector;

public class STV {
    public static Vector vector = new Vector (5,5);
}

class VectorClass {
    void method(Object o) {
        STV.vector.add(o);
    }
}
```

Repair

If a static variable is necessary, set a maximum size and make sure that Vector does not exceed the limit.

```
package GC;
import java.util.Vector;

public class STV {
    public static void addVector(Object o) {
        // checks size of the Vector before calling 'add()'.
        if(vector.size() < MAX) {
            vector.add(o);
        }
    }
    void method(Object o) {
       addVector(o);
    }
    public static Vector vector = new Vector (5,5);
    public static final int MAX = 5;
}
```

GLOBAL.DPAC

Declare package-private classes as inaccessible as possible

Description

This rule flags any package-private class that is more accessible than is necessary.

Making the fields/methods/classes as inaccessible as possible makes the code much more object-oriented and it is much easier to understand code dependencies.

Example

See examples/static/GLOBAL/too-accessible.

Repair

Change the class's accessibility or document the reason for the excessive accessibility.

GLOBAL.DPAF

Declare package-private fields as inaccessible as possible

Description

This rule flags any package-private field that is more accessible than is necessary.

Using Global Static Analysis, Jtest can detect if non-”private” fields/methods/classes can be made less accessible.

Making the fields/methods/classes as inaccessible as possible makes the code much more object-oriented and it is much easier to understand code dependencies.

Example

See examples/static/GLOBAL/too-accessible.

Repair

Change the field’s accessibility or document the reason for the excessive accessibility.

GLOBAL.DPAM

Declare package-private methods as inaccessible as possible

Description

This rule flags any package-private method that is more accessible than is necessary.

Using Global Static Analysis, Jtest can detect if non-”private” fields/methods/classes can be made less accessible.

Making the fields/methods/classes as inaccessible as possible makes the code much more object-oriented and it is much easier to understand code dependencies.

Example

See examples/static/GLOBAL/too-accessible.

Repair

Change the method’s accessibility or document the reason for the excessive accessibility.

GLOBAL.DPPC

Declare “public”/”protected” classes as inaccessible as possible

Description

This rule flags any “public”/”protected” class that is more accessible than is necessary.

Using Global Static Analysis, Jtest can detect if “public”/”protected” fields/methods/classes can be made less accessible.

Making the fields/methods/classes as inaccessible as possible makes the code much more object-oriented and it is much easier to understand code dependencies.

Example

See examples/static/GLOBAL/too-accessible.

Repair

Change the class’s accessibility or document the reason for the excessive accessibility.

GLOBAL.DPPF

Declare “public”/”protected” fields as inaccessible as possible

Description

This rule flags any “public”/”protected” field that is more accessible than necessary.

Using Global Static Analysis, Jtest can detect if non-”private” fields/methods/classes can be made less accessible.

Making the fields/methods/classes as inaccessible as possible makes the code much more object-oriented and it is much easier to understand code dependencies.

Example

See examples/static/GLOBAL/too-accessible.

Repair

Change the field’s accessibility or document the reason for the excessive accessibility.

GLOBAL.DPPM

Declare public/protected methods as inaccessible as possible

Description

This rule flags any “public”/”protected” method that is more accessible than necessary.

Using Global Static Analysis, Jtest can detect if “public”/”protected” fields/methods/classes can be made less accessible.

Making the fields/methods/classes as inaccessible as possible makes the code much more object-oriented and it is much easier to understand code dependencies.

Example

See examples/static/GLOBAL/too-accessible.

Repair

Change the method’s accessibility or document the reason for the excessive accessibility.

GLOBAL.SPAC

Non-subclassed package-private classes should be declared "final"

Description

This rule flags non-subclassed package-private classes that are not declared "final".

There are two advantages to declaring these classes “final”:

- It optimizes the code. The compiler knows that nobody can extend the class or override its methods, so it can generate optimized code.
- It makes the code self documented. Somebody looking at the class will know that no other class extends this class.

Repair

Add the keyword "final" or document the reason for the class not being final.

GLOBAL.SPAM

Non-overridden package-private methods should be declared "final"

Description

This rule flags non-overridden package-private methods that are not declared "final".

There are two advantages to declaring these methods “final”:

- It optimizes the code. The compiler knows that nobody can override the method, so it can it can generate optimized code.
- It makes the code self documented. Somebody looking at the method will know that no other class extends this class.

Repair

Add the keyword "final" or document the reason for the method not being final.

GLOBAL.SPPC

Non-subclassed "public/protected" classes should be declared "final"

Description

This rule flags non-subclassed “public/protected” classes that are not declared "final".

There are two advantages to declaring these classes “final”:

- It optimizes the code. The compiler knows that nobody can extend the class or override its methods, so it can generate optimized code.
- It makes the code self documented. Somebody looking at the class will know that no other class extends this class.

Repair

Add the keyword "final" or document the reason for the class not being final.

GLOBAL.SPPM

Non-overridden "public/protected" methods should be declared "final"

Description

This rule flags non-overridden “public/protected” classes that are not declared "final".

There are two advantages to declaring these methods “final”:

- It optimizes the code. The compiler knows that nobody can override the method, so it can it can generate optimized code.
- It makes the code self documented. Somebody looking at the method will know that no other class extends this class.

Repair

Add the keyword "final" or document the reason for the method not being final.

GLOBAL.UPAC

Avoid globally unused package-private classes

Description

This rule flags any globally unused package-private class.

Using Global Static Analysis, Jtest can detect if non-”private” fields/methods/classes are not used by any class.

These unused entities usually point to either:

1. Old code that is no longer needed and which makes the class more difficult to understand.
2. A logical flaw if the entity needs to be used, but other classes incorrectly avoid using it.

Example

See examples/static/GLOBAL/unused.

Repair

Remove the unused field/method/class or document the reason for its existence.

GLOBAL.UPAF

Avoid globally unused package-private fields

Description

This rule flags any unused package-private field.

Using Global Static Analysis, Jtest can detect if non-”private” fields/methods/classes are not used by any class.

These unused entities usually point to either:

1. Old code that is no longer needed and which makes the class more difficult to understand.
2. A logical flaw if the entity needs to be used, but other classes incorrectly avoid using it.

Example

See examples/static/GLOBAL/unused.

Repair

Remove the unused field/method/class or document the reason for its existence.

GLOBAL.UPAM

Avoid globally unused package-private methods

Description

This rule flags any globally unused package-private method.

Using Global Static Analysis, Jtest can detect if non-”private” fields/methods/classes are not used by any class.

These unused entities usually point to either:

1. Old code that is no longer needed and which makes the class more difficult to understand.
2. A logical flaw if the entity needs to be used, but other classes incorrectly avoid using it.

Example

See examples/static/GLOBAL/unused.

Repair

Remove the unused field/method/class or document the reason for its existence.

GLOBAL.UPPC

Avoid globally unused “public”/”protected” classes

Description

This rule flags any globally unused “public”/”protected” class.

Using Global Static Analysis, Jtest can detect if “public”/”protected” fields/methods/classes are not used by any class.

These unused entities usually point to either:

1. Old code that is no longer needed and which makes the class more difficult to understand.
2. A logical flaw if the entity needs to be used, but other classes incorrectly avoid using it.

Example

See examples/static/GLOBAL/unused.

Repair

Remove the unused field/method/class or document the reason for its existence.

GLOBAL.UPPF

Avoid globally unused public/protected fields

Description

This rule flags any globally unused “public”/”protected” field.

Using Global Static Analysis, Jtest can detect if “public”/”protected” fields/methods/classes are not used by any class.

These unused entities usually point to either:

1. Old code that is no longer needed and which makes the class more difficult to understand.
2. A logical flaw if the entity needs to be used, but other classes incorrectly avoid using it.

Example

See examples/static/GLOBAL/unused.

Repair

Remove the unused field/method/class or document the reason for its existence.

GLOBAL.UPPM

Avoid globally unused “public”/”protected” methods

Description

This rule flags any globally unused “public”/”protected” method.

Using Global Static Analysis, Jtest can detect if “public”/”protected” fields/methods/classes are not used by any class.

These unused entities usually point to either:

1. Old code that is no longer needed and which makes the class more difficult to understand.
2. A logical flaw if the entity needs to be used, but other classes incorrectly avoid using it.

Example

See examples/static/GLOBAL/unused.

Repair

Remove the unused field/method/class or document the reason for its existence.

INIT.CSI

The constructor should explicitly initialize all fields

Description

This rule flags any constructor that does not explicitly initialize all fields.

Example

```
package INIT;

class CSI {
    CSI () {
        this (12);
        k = 0;
    }

    CSI (int val) {
        j = val;
    }

    private int i = 5;
    private int j;
    private int k;
}
```

Repair

In some cases, these errors can be corrected by initializing the appropriate fields. In other cases, these fields will have been assigned values from other uninitialized fields that must be initialized to eliminate the problem.

INIT.NFS

Avoid using non-final "static" fields during the initialization

Description

This rule flags non-final “static” fields that are used during a field's initialization.

Example

```
package INIT;
public class NFS {
    static int max = 10;
    static int count = max; // violation
    int size = NFS.max;    // violation
}
```

Repair

```
package INIT;
public class NFS {
    static final int MAX = 10; //declare a constant.
    static int max = 10;
    static int count = MAX;
    int size = NFS.MAX;
}
```

INIT.INITLV

Initialize all local variables explicitly

Description

This rule flags local variables that are not initialized in the declaration.

Example

```
package INIT;
public class LV {
    private method (int size) {
        int max;    // violation
        int count = size;
        for (int i = 0; i < count; i++ ) {
         // do something.
        }
    }
}
```

Repair

Initialize all local variables in declaration.

INIT.SF

Explicitly initialize all “static” fields

Description

This rule flags any uninitialized static field in your methods.

Example

```
package INIT;

class SF
{
    SF () {}

    static private int K;
    static private int L;  // 'L' is not initialized

    static {
        K = 10;
    }
}
```

Repair

These errors can be corrected by initializing the appropriate static field.

INTER.CLO

A single character should not be prefixed or followed by a logic operator in an internationalized environment

Description

This rule flags code where a single character is prefixed or followed by a logic operator.

If code contains a single character prefixed or followed by a logic operator, it will not run in an internationalized environment.

Example

```
package INTER;
public class CLO{
  public boolean cmpchar(char ch){
        if (( ch >='a' && ch <='z') || (ch >='A' && ch<='Z'))  //
violation
                return true;
          return false;
        }
   }
```

Repair

Use the Character comparison methods, which use the Unicode standard to identify character properties. Thus replace the "if" statement above with: ...

```
if (Character.isLetter(ch))
                          ...
```

For more information on the Character comparison methods, see the section 'Checking Character Properties' at http://java.sun.com/docs/books/tutorial/i18n/text/charintro.html

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

INTER.COS

String concatenation will not work in an internationalized environment

Description

This rule flags code that contains concatenated strings.

For code to be able to run in an internationalized environment, concatenated strings must not be used. This includes using the '.concat()' method with strings or using '+' or '+=' operators with strings. The reason for this is compound messages contain variable data and are difficult to translate. For example the message : "At 12:30 P.M on Jul 3, 2053, there was a disturbance in the Force."

Example

```
import java.util.*;
package INTER;
public class COS{
  public void addstrings(){
      Date current_date = new Date(System.currentTimeMillis());
      String message = "At " + current_date + " there was a
         disturbance in the Force ";    // violation
      String message2 = "on planet 7";
      message = message.concat(message2);   // violation
      System.out.println(message);
  }
}
```

Repair

Do not use '.concat()' or "+", '+=" operations with Strings. Instead, use the "MessageFormat" class to deal with Compound Messages as follows:

```
Object [ ] arguments = {
    new Integer(7),
    new Date(System.currentTimeMillis()),
     "a disturbance in the Force"
};
String message = MessageFormat.format( "At  (1,time) on ( 1, date) ,
there was
{2} on planet { 0,number,integer).", arguments);
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

INTER.DTS

A Date or Time variable should not be followed by 'toString()' in an internationalized environment

Description

This rule flags date or time variables that are followed by ‘toString()’.

For code to be able to run in an internationalized environment, a date variable cannot be followed by '.toString()' because date and time formats differ with region and language.

Example

```
package INTER;

import java.util.*;
import java.awt.*;

public class DTS{
  public void displaydate(){
   Date today = new Date();
   String dateOut = today.toString();  //violation
   System.out.println(dateOut);
   }
}
```

Repair

If you use the date-formatting classes, your application can display dates and times correctly around the world. The "DateFormat" class provides predefined formatting styles that are locale-specific and easy to use. The following code is an example of how to use the "DateFormat" class.

```
Date today;
String dateOut;
DateFormat dateFormatter;
dateFormatter = DateFormat.getDateInstance(DateFormat.DEFAULT,cur-
rentLocale);

today = new Date();
dateOut = dateFormatter.format(today);
System.out.println(dateOut + " " + currentLocale.toString());
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

http://java.sun.com/docs/books/tutorial/i18n/format/dateFormat.html

INTER.NCL

Single character literals should only be placed in constants in an internationalized environment

Description

This rule flags single character literals in non-constants.

If code contains single character literals that are used in non-constants, it will not run in an internationalized environment. This means single character literals can only be used when declared as "static" "final" "char".

Example

```
package INTER;
public class NCL{
 public void Echo() {
  System.out.println('c');  //violation
  }
}
```

Repair

```
package Inter;
public class NCL{
  public void Echo(){
    System.out.println(mychar);
   }
}
static final char mychar ='c';
}
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/

INTER.NSL

String literals should only be placed in constants in an internationalized environment

Description

This rule flags string literals that are not placed in constants.

If code contains single character literals that are not used in constants, it will not run in an internationalized environment. This means string literals can only be used when declared as "static" "final" "String".

Example

```
package INTER;

public class NSL{
   public void Echo () {
      System.out.println("hello");    //Violation
   }
}
```

Repair

```
package Inter;

public class NSL{
  public void Echo(){
   System.out.println(string1);
 }
 static final String string1="hello";
}
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/

INTER.NTS

A number variable should not be followed by 'toString()' in an internationalized environment

Description

This rule flags number variables that are followed by ‘toString()’.

If code contains number variables that are followed by ‘toString()’, it will not run in an internationalized environment because it will not be displayed correctly in all countries. If code displays numbers and currencies, they must be formatted in a locale-independent manner.

Example

```
package INTER;
import java.awt.*;
public class NTS{
  public void displaynum(){
     Double amount = new Double(3.2);
     String displayAmount = amount.toString();  //violation.
     System.out.println(displayAmount);
  }
}
```

Repair

Replace the preceding code with a routine that formats the number correctly. The Java programming language provides several classes that format numbers. One way to format numbers is by using predefined formats. This can be done by using the "NumberFormat" class which allows you to format numbers, currencies, and percentages according to Locale. The following code is an example of formatting a Double according to Locale.

```
Double amount = new Double(3.2);
NumberFormat numberFormatter;
String amountOut;

numberFormatter = NumberFormat.getNumberInstance(currentLo-
cale);
amountOut = numberFormatter.format(amount);
System.out.println(amountOut + "  " + currentLo-
cale.toString());
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

http://java.sun.com/docs/books/tutorial/i18n/format/numberFormat.html

INTER.SB

StringBuffer should not be used in an internationalized environment

Description

This rule flags code that contains ‘StringBuffer’

If code contains ‘StringBuffer’, it will not run in an internationalized environment.

Example

```
package INTER;
public class SB{
  public void mysb(){
    StringBuffer sb= new StringBuffer("hello"); // violation
    sb=sb.append(" gina");
   }
}
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

INTER.SCT

The 'compareTo()' method of strings should not be used in an internationalized environment

Description

This rule flags code that uses the ‘compareTo()’ method of strings.

If code uses the ‘compareTo()’ method of strings, it will not run in an internationalized environment. This is because it performs binary comparisons of the Unicode characters within the two strings, which are ineffective when sorting in most languages. String variables with 'compareTo()' cannot be relied on to sort strings because the Unicode values of the characters in the strings do not correspond to the relative order of the characters for most languages.

Example

```
package INTER;
package INTER;
public class SCT{
  public boolean compstr(){
  String s1= new String("hello");
  String s2 = new String("bye");
  if (s1.compareTo(s2) < 0 )   // violation
     return true;
  return false;
  }
}
```

Repair

The predefined collation rules provided by the "Collator" class should be used instead to sort strings in a locale-independent manner. To instantiate the "Collator" class, invoke the getInstance method. Usually, you create a Collator for the default Locale, as in the following example:

```
Collator myCollator = Collator.getInstance();
```

You can also specify a particular Locale when you create a Collator, as follows:

```
Collator myFrenchCollator = Collator.getInstance(Locale.FRENCH);
```

Then you invoke the Collator.compare method to perform a locale-independent string comparison as follows:

```
myCollator.compare(s1,s2);
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

http://java.sun.com/docs/books/tutorial/i18n/text/locale.html

INTER.SE

The 'equals()' method of strings should not be used in an internationalized environment

Description

This rule flags code that uses the ‘equals()’ method of strings.

If code uses the ‘equals()’ method of strings, it will not run in an internationalized environment because it performs binary comparisons of the Unicode characters within the two strings, which are ineffective when sorting in most languages.

A string variable with '.equals()' cannot be relied on to sort strings. This is because the Unicode values of the characters in the strings do not correspond to the relative order of the characters for most languages.

Example

```
package INTER;

public class SE{
    public boolean eqstr(){
        String s1= new String("hello");
        String s2 = new String("bye");
        if (s1.equals(s2)) // violation
            return true;
        return false;
    }
}
```

Repair

The predefined collation rules provided by the "Collator" class should be used instead to sort strings in a locale-independent manner. To instantiate the "Collator" class, invoke the getInstance method. Usually, you create a Collator for the default Locale, as in the following example :

```
Collator myCollator = Collator.getInstance();
```

You can also specify a particular Locale when you create a Collator, as follows:

```
Collator myFrenchCollator = Collator.getInstance(Locale.FRENCH);
```

Then you invoke the Collator.equals method to perform a locale-independent string comparison as follows:

```
boolean is_equal = myCollator.equals(s1,s2);
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

http://java.sun.com/docs/books/tutorial/i18n/text/locale.html

INTER.ST

StringTokenizer should not be used in an internationalized environment

Description

This rule flags code that uses 'StringTokenizer()'.

If code uses 'StringTokenizer()', it will not run in an internationalized environment.

Example

```
package INTER;
public class ST{
 public void myst(String str){
  StringTokenizer st = new StringTokenizer(str);  // violation
   int i=st.countTokens();
   }
}
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

INTER.TTS

A Time variable should not be followed by 'toString()' in an Internationalized environment

Description

This rule flags code where a Time variable is followed by ‘toString()’

If code contains a Time variable followed by a ‘toString()’, it will not run in an internationalized environment because time formats differ with region and language.

Example

```
package INTER;
import java.sql.*;
import java.awt.*;
public class TTS(){
  public void displaytime() {
   Time t1= new Time(1000);
   String timeString= t1.toString();    //violation
   System.out.println(timeString);
  }
}
```

Repair

If you use the date-formatting classes, your application can display dates and time correctly around the world. First, you create a formatter with the

getTimeInstance method as follows:

```
DateFormat timeFormatter = DateFormat.getTimeInstance(
          DateFormat.DEFAULT, currentLocale);
```

Second, you invoke the format method, which returns a String containing the formatted date/time as follows:

```
Time t1 = new Time(1000):
String timeString= timeFormatter.format(t1);
```

Reference

http://java.sun.com/docs/books/tutorial/i18n/intro/checklist.html

http://java.sun.com/docs/books/tutorial/i18n/format/dateFormat.html

JAVADOC.BT

Bad tag in Javadoc comment

Description

This rule flags any bad tag in your code's Javadoc comments.

A Javadoc comment may use special tags which all begin with the @ character and allow Javadoc comments to provide additional formatting for the documentation. Check the HTML markup tags which the Javadoc comments may contain.

Example

```
/**
 * This is a comment
 * @unsupported tag
 */

package JAVADOC;

public class BT {

}
```

Repair

Use only supported tags.

JAVADOC.MAJDT

Missing ’@author’ Javadoc tag in a "class" Javadoc comment

Description

This rule flags any Javadoc comment that does not have an @author tag.

Each class and interface’s Javadoc comment should have an @author Javadoc tag.

Example

```
/**
  * @version:
  * @see:
  */
   // violation, Class’s Javadoc does not have
    // "@author" tag.
package JAVADOC;
public class MAJDT {
    // some declaration.
}
```

Repair

Provide an “@author” tag in the Javadoc comment.

JAVADOC.MJDC

Missing Javadoc comments

Description

This rule flags any class, method, or field that does not have a Javadoc comment block.

Every class, interface, method, and field should have a Javadoc comment block.

Example

```
package JAVADOC;

public class MJDC {   // violation, no Javadoc comments for this
        class.
    /** @param:  size
      */
    public MJDC (int size) {}
}
```

Repair

Provide Javadoc comments for each "class", "interface", method, and field.

JAVADOC.MVJDT

Javadoc comments for classes and interfaces should have ’@version’ Javadoc tag

Description

This rule flags any class or interface Javadoc comment that does not have an @version Javadoc tag.

Each class and interface’s Javadoc comment should have an @version Javadoc tag.

Example

```
/**
  * @author:
  * @see:
  */
   // violation, Class’s Javadoc does not have
    //"@version" tag.
package JAVADOC;
public class MVJDT {
    // some declaration.
}
```

Repair

Provide an “@version” tag in the Javadoc comment.

JAVADOC.PARAM

Method should have equal number of '@param' Javadoc tags as parameters

Description

This rule flags any method argument that does not have a corresponding ‘@param’ Javadoc tag.

Example

```
package JAVADOC;
public class PARAM {
    /**
     */   //violation
    private void setId (String name) {
        _id = name;
    }
    String _id;
}
```

Repair

Provide an @param tag for each method that has arguments.

METRICS.CIHL

"class" or "Interface" inheritance level

Description

This metric measures “class” or “interface” inheritance level.

An unnecessarily deep class hierarchy adds to complexity and can represent a poor use of the inheritance mechanism.

By default, Jtest reports an error if this metric’s value is not between 0 and 5.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.CTNL

Number of lines in "class" or "interface"

Description

This metric measures the number of lines of code in each “class” or “interface”.

This metric can be used to estimate application size. A class should be less than 1000 lines long.

By default, Jtest reports an error if this metric’s value is not between 0 and 1000.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NOF

Number of fields

Description

This metric measures the number of fields in each class.

The number of fields in a class indicates the amount of data that the class must maintain in order to carry out its responsibilities.

By default, Jtest reports an error if this metric's value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NOM

Number of methods

Description

This metric measures the number of methods in each class.

The number of methods per class indicates the total level of functionality implemented by a class.

By default, Jtest reports an error if this metric's value is not between 0 and 20.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.PJDC

Percentage of Javadoc comments(%)

Description

This metric measures the percentage of Javadoc comments in a class or interface.

This metric is measured using the following formula:

% javadoc = #Javadoc in a file / (#fields + #methods + 1) * 100

Note: 1 is for either the class or interface's Javadoc comments.

Every class, interface, method, and field should have a Javadoc comment associated with it.

By default, Jtest reports an error if this metric's value is not between 60% and 100%.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPKGF

Number of package-private fields

Description

This metric measures the number of package-private fields in each class.

The number of fields in a class indicates the amount of data that the class must maintain in order to carry out its responsibilities.

By default, Jtest reports an error if this metric's value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPKGM

Number of package-private methods

Description

This metric measures the number of package-private methods per class.

The number of methods per class indicates the total level of functionality implemented by a class.

By default, Jtest reports an error if this metric's value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPRIF

Number of "private" fields

Description

This metric measures the number of “private” fields in a class.

The number of fields in a class indicates the amount of data the class must maintain in order to carry out its responsibilities.

By default, Jtest reports an error if this metric’s value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPRIM

Number of "private" methods

Description

This metric measures the number of “private” methods per class.

The number of methods per class indicates the total level of functionality implemented by a class.

By default, Jtest reports an error if this metric’s value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPROF

Number of "protected" fields

Description

This metric measures the number of “protected” fields in a class.

The number of fields in a class indicates the amount of data that the class must maintain in order to carry out its responsibilities.

By default, Jtest reports an error if this metric’s value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPROM

Number of "protected" methods

Description

This metric measures the number of “protected” methods in a class.

The number of methods per class indicates the total level of functionality implemented by a class.

By default, Jtest reports an error if this metric’s value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPUBF

Number of "public" fields

Description

This metric measures the number of “public” fields in a class.

The number of fields in a class indicates the amount of data that the class must maintain in order to carry out its responsibilities.

By default, Jtest reports an error if this metric’s value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.NPUBM

Number of "public" methods

Description

This metric measures the number of “public” methods per class.

The number of methods per class indicates the total level of functionality implemented by a class.

By default, Jtest reports an error if this metric’s value is not between 0 and 10.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.STMT

Number of statements in a method

Description

This metric measures the number of statements in a method.

By default, Jtest reports an error if this metric;s value is not between 0 and 20.

Repair

Break the method into several methods.

METRICS.TCC

Cyclomatic Complexity

Description

This metric measures each method's cyclomatic complexity by measuring the number of "while", "for", "if", and "switch" statements in a method.

Cyclomatic complexity measures method complexity.

Studies have found that methods with cyclomatic complexity higher than 10 are more error-prone than methods with lower cyclomatic complexity.

By default, Jtest reports an error if this metric's value is not between 0 and 10.

Note: This rule does not test abstract methods, native methods, or any methods that are declared in the interface.

Reference

Grady, Robert B. *Practical Software Metrics For Project Management and Process Improvement*. Prentice Hall P T R, 1992, pp.16 - 18.

METRICS.TNLM

Number of lines in a method

Description

This metric measures the number of lines of code in a method.

The number of lines of code in a method estimates method length and complexity.

By default, Jtest reports an error if this metric's value is not between 0 and 30.

Note: This rule does not test abstract methods, native methods, or any methods that are declared in the interface.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.TNMC

Number of method calls

Description

This metric measures the number of method calls to methods and system functions within a system, class, or method.

By default, Jtest reports an error if this metric's value is not between 0 and 20.

Note: This rule does not test abstract methods, native methods, or any methods that are declared in the interface.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.TNOP

Number of parameters

Description

This metric measures the number of parameters in a class.

A high number of parameters indicates a complex interface to calling objects.

By default, Jtest reports an error if this metric's value is not between 0 and 5.

Reference

Mark Schroeder. "A Practical Guide to Object-Oriented Metrics." *IT Pro*, November/December, 1999.

METRICS.TRET

Number of "return" statements

Description

This metric measures the number of “return” statements in a class.

Methods with multiple return points are easy to program, but they are more difficult to debug and maintain.

By default, Jtest reports an error if this metric’s value is not between 0 and 1.

Note: This rule only checks explicit "return" statements. It does not test abstract or native methods or any methods that are declared in the interface.

MISC.AFP

Avoid assignment to method parameters

Description

This rule flags any assignment to method parameters.

Assignment operations on a method parameter can cause problems when the value of the parameter is used more than once in a block.

Example

```
package MISC;

class AFP {
    int avg (int x) {
        int count = 0;
        while (x++ < 10) {
        count += x;
        }
    return count % x; // x no longer has the same value.
    }
}
```

Repair

```
//create a local variable.
class AFP {
    int avg (int x) {
        int count = 0;
        int i = x;
        while (i++ < 10) {
            count += i;
        }
        return count % i;
    }
}
```

MISC.ASFI

A class with only "abstract" methods and "static", "final" fields should be declared as an "interface"

Description

This rule flags classes that should be declared as “interface”s.

"abstract" classes that only contain method signatures and "static", "final" fields should be declared as "interface" because they are effectively the same. A "class" can extend only one "class" but it can implement multiple interfaces.

Example

```
package MISC;

abstract class ASFI {
    abstract void method();
    final static String ID = "MISC_ASFI";
}
```

Repair

Replace "abstract" class to an "interface".

```
package MISC;

interface ASFI {
    void method();
    String ID = "MISC_ASFI";
}
```

Reference

Warren, Nigel, and Bishop, Philip. *Java in Practice.* Addison-Wesley, 1999, pp.22-23.

MISC.CLONE

A ‘clone()’ method should invoke ‘super.clone()’ in the body

Description

This rule flags any ‘clone()’ method that does not invoke ‘super.clone()’ in the body.

‘super.clone()’ invokes the method Object.clone, which creates an object of the correct type. Object.clone initializes each field in the new clone object by assigning it the value from the same field of the object being cloned. Therefore, if ‘super.clone()’ is not called, the object might not be initialized correctly.

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp. 77 - 82.

MISC.CTOR

Use care when calling non-”final” methods from constructors

Description

This rule flags code that calls a non-”final” method from a constructor.

The purpose of the constructor is to initialize an object. It might call some methods of its class. If it calls a non-”final” method, a derived class can override the method, depending upon how the overridden method is coded. This can lead to unexpected results.

Example

```
package MISC;
public class CTOR {
    public CTOR() {
        _size = readSize(); // violation, it can be overridden by
                                    // CTOR’s derived class
    }
    public int readSize() {
        return fis.read();
    }
    private FileInputStream fis = new FileInputStream ("data.out");
    private int _size;
}
```

Repair

If a constructor needs to call a method for an initialization, make this method final, or private. Or, if a method is the part of some package, create a private void init() method to do all the initialization.

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.238 - 240.

MISC.EFB

Each “for” statement should have a block

Description

This rule flags any “for” statement that does not have a block.

“for” statements are less error-prone when they have blocks.

Example

```
package MISC;

public class EFB {
    int i = 3;

    void method () {
        for (int i = 0; i < 10; i++)
            System.out.println (i * i);
            System.out.println (i);
    }
}
```

Repair

Provide a block for each “for” statement regardless of the number of statements intended to be executed within the block.

MISC.ELSEBLK

"else" statements should have a block

Description

This rule flags any “else” statement that does not have a block.

“else” statements are less error-prone when they have blocks.

Example

```
package MISC;

public class ELSEBLK {
    public void method (boolean b) {
        if (b) {
            System.out.println ("inside of if");
        } else
            System.out.println ("inside of if");  //violation.
    }
}
```

Repair

Provide a block for each “else” statement.

MISC.FF

Constant fields should be declared as "final"

Description

This rule flags constant fields that are not declared as “final”.

Fields that are initialized in a declaration and values that do not change throughout the class should be "final" fields.

Example

```
package MISC;
public class FF {
    private int size = 5;  // violation
    private void method () {
        // do something but size's value do not change.
    }
}
```

Repair

Declare 'size' "final".

MISC.FLV

Constant local variables should be declared as "final"

Description

This rule flags constant local variables that are not declared as “final”.

Local variables that are initialized in a declaration whose values do not change throughout the block should be declared "final".

Example

```
package MISC;
public class FLV {
    private void method () {
          int size = 0;   //violation
        // do something but size's value do not change.
    }
}
```

Repair

Make 'size' as a "final".

MISC.HMF

Avoid hiding member fields in member methods

Description

This rule flags code where a member fields is hidden in a member method.

Variables declared local to a “class” method can hide instance fields of the same name, effectively blocking access to the field.

Local variables with the same name as instance fields can make your code difficult to read. It is good practice to make a variable’s name as unique as possible for coding clarity

Example

```
package MISC;

abstract class HMF {
    public int method2 () {
        int i = 5;       // violation
        System.out.println (i);
    }

    private int i = 0;
}
```

Repair

Give the local variable a unique name.

MISC.IFBLK

"if" statements should have a block

Description

This rule flags “if” statements that do not have a block.

"if" statements are less error-prone if they have a block.

Example

```
package MISC;

public class IFBLK {
    public void method (boolean b) {
        if (b)    // violation
            System.out.println ("inside of if");
        if (b) {
            System.out.println ("inside of if");
        }
    }
}
```

Repair

Provide a block for each "if" statements.

MISC.CLNC

Avoid using a constructor for implementing 'clone()'

Description

This rule flags code that uses a constructor for implementing ‘clone()’.

Using a constructor when implementing the 'clone()' method restricts sub-classes from reusing the 'clone()' method of the superclass.

Example

```
package MISC;

public class CLNC {
    public Object clone() {
        CLNC cl = new CLNC();    // violation
        // get attributes' clone.
        return cl;
    }
}
```

Repair

Use 'super.clone()' instead of calling a constructor to implement 'clone()'.

```
public class CLNC {
    public Object clone() {
        CLNC cl = (CLNC) super.clone();
        // get clone for cl's attributes.
        return cl;
    }
}
```

Reference

Daconta, M, Monk, E, Keller, J, and Bohnenberger, K. *Java Pitfalls*. John Wiley & Sons, 2000, pp.12 - 14.

MISC.MSF

Avoid too many "static" fields

Description

This rule flags code that contains more than two “static” fields.

Static variables act global in non-OO languages. They make methods more context-dependent, hide possible side-effects, sometimes present synchronized access problems, and are the source of fragile, non-extensible constructions. Also, neither static variables nor methods can be overridden in any useful sense in subclasses.

Example

```
package MISC;

public class MSF {
    private static int s_size;
    static String s_title;        // violation

    static void setFields (int size, String title) {
        s_size = size;
        s_title = title;
    }
}
```

Repair

Try to minimize static fields if possible.

Reference

http://g.oswego.edu/dl/html/javaCodingStd.html

MISC.PCF

Provide a condition in a “for” loop

Description

This rule flags any “for” loop that does not have condition.

Failing to provide a condition in a “for” loop might cause infinite loops.

Example

The following example shows a “for” statement that has an empty second slot. This might cause infinite loops.

```
package MISC;

public class PCF {
    void method () {
        int i = 0;
        int j = 0;
        for (;; i++) {
            j++;
            break;
        }
    }
}
```

Repair

Consider providing a condition that will be tested before each new pass through the loop.

MISC.PIF

Provide an incremental segment/part in a “for” loop or use a “while” loop

Description

This rule flags any “for” statement that does not have an incremental part.

This may be an indication that the incremental part is missing or that the code would be clearer if a “while” statement were used instead of a “for” statement.

Example

```
package MISC;

public class PIF {
    void method () {
        int i = 0;
        for (; i < 1000; ) {
            if (i % 2 == 0) {
                i = 2 * i + 1;
            } else {
                i = i/2;
            }
        }
    }
}
```

Repair

Check if the third argument of the “for” statement is missing. If it is missing, either increment the counter within the “for” structure or change the “for” statement to a “while” statement.

MISC.WHILE

A "while" statement should have a block

Description

This rule flags any “while” statement that does not have a block.

It is a good programming practice to add optional block for a "while" statements.

Example

```
package MISC;

public class WHILE {
    public void method (int count) {
        while (count++ < 10)    // violation
            System.out.println ("inside of while");
        while while (count-- > 0) {
            System.out.println ("inside of while");
        }
    }
}
```

Repair

Add a block for each "while" statements.

NAMING.CVN

Use conventional variable names

Description

This rule flags any variable that has an unconventional name format.

Use conventional variable names for one-character names.

- b for a byte
- c for a char
- d for a double
- f for a float
- i, j, and k for integers
- l for a long
- o for an Object
- s for a String
- v for an arbitrary value of some type

Example

```
package NAMING;

public class CVN {
    void method () {
        int b = 1;
        int d = 1;
    }
}
```

Repair

Use conventional variable names.

NAMING.GETA

Accessor method names should start with ’get’

Description

This rule flags any accessor method that has an unconventional name format.

A getter method’s name should start with ’get’ unless it returns boolean. If it returns boolean, it should be prefixed by ’is’.

Example

```
package NAMING;

public class GETA {
    public int method() {     // violation.
        return _count;
    }
    private int _count = 0;
}
```

Repair

Change accessor method name to ’get<field name>’.

```
package NAMING;
public class GETA {
    public int getCount() {
        return _count;
    }
    private int _count = 0;
}
```

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.GETB

Getter methods name should start with ’is’

Description

This rule flags any getter method that has an unconventional name format.

Names of getter methods that return “boolean” values should start with ’is’.

Example

```
package NAMING;
public class GETB {
    public boolean method() {     // violation.
        return _ready;
    }
    private boolean _ready = false;
}
```

Repair

Change accessor method name to is<field name>.

```
package NAMING;
public class GETB {
    public boolean isReady() {
        return _ready;
    }
    private boolean _ready = false;
}
```

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.IFV

Use uppercase letters for “interface” fields

Description

This rule flags any “interface” field that has an unconventional name format.

Fields in an “interface” are always “static” and “final”, so they should follow the same naming convention for named constants.

Example

```
package NAMING;

interface IFV {
    int max = 1000;
}
```

Repair

```
interface IFV {
    int MAX = 1000;
}
```

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp. 91-93.

NAMING.IRB

Use ‘is...’ for naming methods that return a “boolean”

Description

This rule flags any function whose name begin with ‘is’ that does not return a ‘boolean’.

Example

```
package NAMING;

public class IRB {
   public static boolean isOK() {
      return true;
   }
   public int isNotOK() {         // Violation
      return 1;
   }
   public int numberOK() {
      return 1;
   }
}
```

Repair

Use standard naming conventions for legibility.

NAMING.NCL

Enforce name format of classes

Description

This rule flags any class that has an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for classes. Jtest will enforce the name format of the “class” using a regular expression.

Example

```
package NAMING;

public class NCL {
}
```

Repair

Capitalize the first letter of the “class" name and use mixed case for the rest of the "class" name.

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.NE

Enforce name format of exceptions

Description

This rule flags any exception that has an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for exceptions. Jtest will enforce the name format of the exception using a regular expression.

Example

```
package NAMING;

public class NE extends RuntimeException {
}
```

Repair

Change the name of the exception to match the regular expression.

NAMING.NIF

Enforce name format of non-”static” fields

Description

This rule flags any non-"static" field with an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for non-"static" fields. Jtest will enforce the name format of the non-"static" fields using a regular expression.

Example

```
package NAMING;

class NIF {
    public int field = 0;
}
```

Repair

Change the name of the non-"static" field to match the regular expression.

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.NITF

Enforce name format of interfaces

Description

This rule flags any interface that has an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for interfaces. Jtest will enforce the name format of the “interface” using a regular expression.

Example

```
package NAMING;

public interface NIFT {
}
```

Repair

Capitalize the first letter of the "interface" name and use mixed case for the rest of the "interface" name.

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.NLV

Enforce name format of local variables

Description

This rule flags any local variable that has an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for local variables. Jtest will enforce the name format of the local variable using a regular expression.

Example

```
package NAMING;

public class NLV {
    void method () {
        int i = 0;
        i++;
    }
}
```

Repair

Change the name of the local variable to match the regular expression. Use complete, descriptive English words.

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.NM

Enforce name format of non-”static” methods

Description

This rule flags any non-”static” method with an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for methods. Jtest will enforce the name format of the non-”static” method using a regular expression.

Example

```
package NAMING;
public class NM {
    void Method () {
    }
}
```

Repair

Change the the non-”static” method’s name to match the regular expression.

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.NMP

Enforce name format of method parameters

Description

This rule flags any method parameter that has an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for method parameters. Jtest will enforce the name format of the method parameter using a regular expression.

Example

```
package NAMING;

public class NMP {
    void method (int i) {
        if (i == 10) return;
    }
}
```

Repair

Change the name of the method parameter to match the regular expression.

NAMING.NSF

Enforce name format of non-“final” “static” fields

Description

This rule flags any “static” field that has an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for “static” fields. Jtest will enforce the name format of the field using a regular expression.

Example

```
package NAMING;

class NSF {
    static int staticField = 0;
}
```

Repair

Change the name of the field to match the regular expression.

NAMING.NSM

Enforce name format of "static" methods

Description

This rule flags any "static" method that has an unconventional name format.

Most programmers or groups of programmers develop a set of naming rules for "static" methods. Jtest will enforce the name format of the "static" method using a regular expression.

Example

```
package NAMING;

class NSM {
    static int staticMethod () {
    }
}
```

Repair

Change the name of the "static" method to match the regular expression.

NAMING.PKG

Use lower case letters for “package" names

Description

This rule flags any “package” name that does not use lower case letters.

Package names should begin with a lower case letter. This prevents the possibility of mistaking a package name for a class name. In fact, package names should usually consist of only lower case letters.

Example

```
package NAMING;    // violation

public class PKG {
}
```

Repair

Change upper case letters to lower case letters.

```
package naming;

public class PKG {
}
```

Reference

Larman, G, Guthrie, R *Java 2 Performance and Idiom Guide*. Prentice Hall, 1999, 248 - 249.

NAMING.SETA

Setter method names should start with ’set’

Description

This rule flags any setter method that has an unconventional name format.

Setter methods’ names should start with ’set’ to follow the JavaBeans naming conventions.

Example

```
package NAMING;

public class SETA {
    public void method(int count) {     // violation.
        _count = count;
    }
    private int _count = 0;
}
```

Repair

Change accessor method name to ’set<field name>’.

```
package NAMING;
public class SETA {
    public void setCount(int count) {     // violation.
        _count = count;
    }
    private int _count = 0;
}
```

Reference

http://www.AmbySoft.com/javaCodingStandards.pdf

NAMING.USF

Use only uppercase letters and underscores when naming “final” “static” fields

Description

This rule flags any “final” “static” field that has an unconventional name format.

Use only uppercase letters and underscores when naming “final” "static" fields (i.e. named constants).

Example

```
package NAMING;

class USF {
    public static final int size = 10;  // Violation
}
```

Repair

```
class USF_fixed {
    public static final int SIZE = 10;
}
```

Reference

Java Language Specification, section 6.8.5.

OOP.AHF

Avoid hiding inherited instance fields

Description

This rule flags any inherited instance variable that is hidden by a member declared in a child class.

Example

```
package OOP;

public class AHV {
    protected long a = 4;
}

public class AHV_ extends AHV {
   protected int a = 5;
}
```

Repair

The solution will depend on the design of the program, but may be as simple as using the inherited member and removing the member declared in the child class.

OOP.AHSM

Avoid hiding inherited “static” member methods

Description

This rule flags any inherited “static” method that is hidden by a child class.

Example

```
package OOP;

public class AHSM {
    static void method1 () {}
}

class AHSM_ extends AHSM {
    static void method1 () {}
}
```

Repair

The solution will depend upon the design of the program, but might be as simple as using the inherited method and removing the method declared in the child class.

OOP.AIC

An inner “class” should only be used if it is going to associate with and be visible to the class that contains it

Description

This rule flags any inner class that is not “private”.

The inner classes automatically have access to their containing classes' member fields.Therefore, problems can arise if an inner class is not a “private” class.

Example

```
package OOP;

class AIC {
    int getSize () {
        return _size;
    }
    class Inner {
        void setSize(int size) {
            _size = size;
        }
    }
    private int _size;
}
```

Repair

```
class AIC {
    int getSize () {
        return _size;
    }
    private class Inner {
```

```
        void setSize(int size) {
           _size = size;
        }
    }
    private int _size;
}
```

Reference

Warren, Nigel, and Bishop, Philip. *Java in Practice* . Addison-Wesley, 1999, pp. 10-11.

OOP.APPF

Avoid “public” or package-private instance fields

Description

This rule flags any “public” or package-private instance field.

Instance fields are an implementation detail of your class. It is good practice to hide such implementation details from users of your class. Instead of making these fields “public” or package-private, provide the user with methods to access and/or change these fields.

Example

```
package OOP;

class APPF {
   int a = 10;           // Violation
   private int c = 14;
}
```

Repair

Declare the field “private” and provide access methods to the field if needed.

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.170 - 173.

OOP.APROF

Avoid “protected” instance fields

Description

This rule flags any “protected” instance field.

Instance fields are an implementation detail of your class. It is good practice to hide such implementation details from users of your class. Instead, provide the user with methods to access and/or change such fields.

Example

```
package OOP;

class APROF {
   private int c = 14;
   protected int a = 10;           // Violation
}
```

Repair

Declare the field “private” and provide access methods to the field if needed.

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.170 - 173.

OOP.IIN

Implement interfaces non-trivially or “abstract”

Description

This rule flags any interface that is implemented trivially or not explicitly declared to be “abstract”.

Use “abstract” methods in base classes rather than those with default no-op implementations. The Java compiler will force subclass authors to implement “abstract” methods, thereby avoiding problems that would have occurred if they forgot to do so when they should have.

Example

```
package OOP;

abstract public class IIN implements B {
    public void f () {
        int i = 9;
        i = 10;
    }

    public void g () {}
    abstract public void h ();
}

interface B {
    public void f ();
    public void g ();
    public void h ();
}
```

Repair

The proper body for the offending method might not have been added, or this method might have been declared unintentionally. In either case, the method should be declared to be “abstract”.

Reference

http://g.oswego.edu/dl/html/javaCodingStd.html

OOP.LEVEL

Avoid more than two levels of nesting classes

Description

This rule flags nesting that is more than two levels deep.

Nesting that is more than two levels deep can be difficult to follow.

Example

```
package OOP;

public class LEVEL {
    class Level1 {
        class Level2 {
            class Level3 {
                private boolean _isClosed = false;
            }
            private int _count = 0;
        }
        private int _size = 0;
    }
    private int _length = 0;
}
```

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp .53.

OOP.LPF

List all “public” and package-private methods/data first

Description

This rule flags code that does not list all “public” and package-private methods/data first.

It is a good habit to order the methods and data in a class so that all “public” and package wide information is presented first, followed by the “protected” and “private” information. This way, the user of your class will find the accessible methods directly underneath the class declaration, and will not be bothered with implementation details described by “private” and “protected” data/methods.

Example

```
package OOP;

class LPF {
    private int method () {
        return 2;
    }

    int method2 () {
        return 3;
    }
}
```

Repair

Reorder the list of methods/data in your class.

OOP.OPM

Do not ‘override’ a private method

Description

This rules flags code where a method overrides a “private” method from a superclass.

Note that a “private” method in a super “class” is not overridden by a method with the same name in the current “class”. This can be confusing. Also, if the method access in the superclass is changed to non-private, the program semantics will change because the method will then be overridden.

Example

```
package OOP;

class OPM extends Super {
    private void method () {}
}

class Super {
    private void method () {}

    public static void main (String[] args) {
        OPM x = new OPM ();
        test (x);
    }

    private static void (Super x) {
        x.method ();  // invokes "Super.method" not "OPM.method"
    }
}
```

Repair

Use different names for the methods to clarify that they are unrelated and to avoid possible problems if the access modifier is changed.

OPT.AAS

Use abbreviated assignment operators

Description

This rule flags code that should use the abbreviated assignment operator, but does not.

In order to write programs more rapidly, you should use the abbreviated assignment operator because doing so causes some compilers run faster.

Example

```
package OPT;
public class AAS {
    void method () {
        int i = 3;
        String s = "fu";
        i = i - 3; //should use “-=”
        s = s + "bar"; //should use “+=”
    }
}
```

Repair

Use the abbreviated assignment operators.

OPT.CEL

Avoid using complex expressions in loop condition statements

Description

This rule flags any complex expression in a loop condition statement.

Unless the compiler optimizes it, the loop condition will be calculated for each iteration over the loop. If the condition value is not going to change, moving the complex expression out of the loop will cause the code to execute faster.

Example

```
package OPT;
import java.util.Vector;
class CEL
{
    void method (Vector vector) {
        for (int i = 0; i < vector.size (); i++)  // Violation
            ; // ...
    }
}
```

Repair

```
class CEL_fixed
{
    void method (Vector vector) {
        int size = vector.size ()
        for (int i = 0; i < size; i++)
            ; // ...
    }
}
```

OPT.CS

Close stream in “finally” blocks

Description

This rule flags code where the last “catch” block of a “try” statement does not contain a “finally” block.

Programs using certain types of resources should release them in order to avoid resource leaks. This can be done through a “finally” block which is placed after the last “catch” block of a “try” statement. Regardless of the outcome of the program, the “finally” block gets executed.

Example

```
import java.io.*;

package OPT;

public class CS {
    public static void main (String args[]) {
        CS cs = new CS ();
        cs.method ();
    }

    public void method () {
        try {
            FileInputStream fis = new FileInputStream ("CS.java");
            int count = 0;
            while (fis.read () != -1)
                count++;
            System.out.println (count);
            fis.close ();
        } catch (FileNotFoundException e1) {
        } catch (IOException e2) {
        }
    }
}
```

Repair

Add a “finally” block after the last “catch”.

Reference

Haggar, Peter. *Practical Java - Programming Language Guide.* Addison Wesley, 2000, pp. 77 -79.

OPT.DIC

Define initial capacities for ‘ArrayList’, ‘HashMap’, ‘HashSet’, ’Hashtable’, ’Vector’, and ‘WeakHashMap’

Description

This rule flags any ‘ArrayList’, ‘HashMap’, ‘HashSet’, ’Hashtable’, ’Vector’, or ‘WeakHashMap’ whose initial capacity is not defined.

Expansion of Vector capacity involves allocating a larger array and copying the contents of the old array to the new one. Eventually, the old array object gets reclaimed by the garbage collector. Vector expansion is an expensive operation. The default 10-element capacity of a Vector is often not enough. Usually, you will be able to approximate the expected size; if so, you should use this value instead of the default.

Example

```
package OPT;

import java.util.Vector;

public class DIC {
    public void addObjects (Object[] o) {
        // if length > 10, Vector needs to expand
        for (int i = 0; i< o.length;i++) {
            v.add(o[i]);   // capacity before it can add more ele-
ments.
        }
    }
    public Vector v = new Vector();  // no initialCapacity.
}
```

Repair

Define initial capacity when known.

```
public Vector v = new Vector(20);
```

References

Bulka, Dov. *Java Performance and Scalability Volume 1: Server-Side Programming Techniques*. Addison Wesley, 2000. pp.55 - 57.

Ford, Neal. “Performance Tuning With Java Technology,” JavaOne 2001 Conference.

OPT.DUN

Don't use the negation operator frequently

Description

This rule flags code that uses the negation operator frequently (i.e., more than two times).

The negation operator (!) decreases the readability of the program.

Example

```
package OPT;

public class DUN {
    boolean method (boolean a, boolean b) {
        if (!a)
            return !a;
        else
            return !b;
    }
}
```

Repair

Do not use the negation operator if possible.

OPT.IF

Use conditional operator for “if (cond) return; else return;” statements

Description

This rule flags “if(cond)-else” statements that could be replaced with “return (cond ? x : y)”.

The conditional operator provides a single expression that yields one of two values based on a boolean expression. The conditional operator results in a more compact expression.

Example

```
package OPT;
public class IF {
    public int method(boolean isDone) {
        if (isDone) {
            return 0;
        } else {
            return 10;
        }
    }
}
```

Repair

If the “if(cond)-else” statements might return a boolean constant, use "return (cond)" instead.

```
package OPT;
public class IF {
    public int method(boolean isDone) {
        return (isDone ? 0 : 10);
    }
}
```

OPT.IFAS

Use conditional assignment operator for “if (cond) a = b; else a = c;” statements

Description

This rule flags any “if(cond)-else” statements that could be replaced by “var = (cond ? x : y)”.

The conditional operator provides a single expression that yields one of two values based on a boolean expression. The conditional operator results in a more compact expression.

Example

```
package OPT;
public class IFAS {
    void method(boolean isTrue) {
        if (isTrue) {
            _value = 0;
        } else {
            _value = 1;
        }
    }
    private int _value = 0;
}
```

Repair

```
package OPT;
public class IFAS {
    void method(boolean isTrue) {
        _value = (isTrue ? 0 : 1); // compact
                                              // expression.
    }
```

```
    private int _value = 0;
}
```

OPT.INSOF

Test "instanceof" to an interface

Description

This rule flags code that performs “instanceof” tests to a class rather than to an interface.

Because interface-based design allows for the flexible inclusion of different implementations, it is generally beneficial. Whenever possible, perform "instanceof" tests to an interface rather than a class.

Example

```
package OPT;

public class INSOF {
    private void method (Object o) {
        if (o instanceof InterfaceBase) { }  // better
        if (o instanceof ClassBase) { }   // worse.
    }
}

class ClassBase {}
interface InterfaceBase {}
```

Reference

Graig Larman, Rhett Guthrie. *Java 2 Performance and Idiom Guide*. Prentice Hall, 2000, p.207.

OPT.IRB

Use ‘System.arraycopy ()’ instead of using a loop to copy an array

Description

This rule flags code where a loop is used to copy arrays.

‘System.arraycopy ()’ is much faster that using a loop to copy an array.

Example

```
package OPT;

public class IRB
{
    void method () {
        int[] array1 = new int [100];
        for (int i = 0; i < array1.length; i++) {
            array1 [i] = i;
        }
        int[] array2 = new int [100];
        for (int i = 0; i < array2.length; i++) {
            array2 [i] = array1 [i];                // Violation
        }
    }
}
```

Repair

```
package OPT;

public class IRB
{
    void method () {
        int[] array1 = new int [100];
        for (int i = 0; i < array1.length; i++) {
            array1 [i] = i;
```

```
        }
        int[] array2 = new int [100];
        System.arraycopy(array1, 0, array2, 0, 100);
    }
}
```

Reference

http://www.cs.cmu.edu/~jch/java/speed.html

OPT.LOOP

Avoid instantiating variables in the loop body

Description

This rule flags any variable that is instantiated in the loop body.

Instantiating temporary Objects in the loop body can increase the memory management overhead.

Example

```
package OPT;
import java.util.Vector;

public class LOOP {
    void method (Vector v) {
        for (int i=0;i < v.isEmpty();i++) {
            Object o = new Object();
            o = v.elementAt(i);
        }
    }
}
```

Repair

Declare a variable outside of the loop and reuse this variable.

```
package OPT;
import java.util.Vector;

public class LOOP {
    void method (Vector v) {
        Object o;
        for (int i=0;i<v.isEmpty();i++) {
            o = v.elementAt(i);
        }
```

```
        }
    }
```

OPT.MAF

Make accessor methods for instance fields “final”

Description

This rule flags non-”final” accessor methods for instance fields.

Accessor methods for instance fields should be made “final”. This tells the compiler that the method cannot be overridden and thus can be inlined.

Example

```
package OPT;

class MAF {
    public void setSize (int size) {
         _size = size;
    }
    private int _size;
}
```

Repair

```
class DAF_fixed {
    final public void setSize (int size) {
         _size = size;
    }
    private int _size;
}
```

Reference

Warren, Nigel. and Bishop, Philip. *Java in Practice.* Addison-Wesley, 1999, pp 4-5.

OPT.PCTS

Use 'charAt()' instead of 'startsWith()' for one character comparisons

Description

This rule flags code that uses ‘startsWith()’ instead of ‘charAt()’ for one character comparisons.

Using 'startsWith()' with a one character argument works fine, but from a performance perspective, this method is a misuse of the String API. 'startsWith()' makes quite a few computations preparing itself to compare its prefix with another string.

Example

```
package OPT;
public class PCTS {
    private void method(String s) {
        if (s.startsWith("a")) { // violation
            // ...
        }
    }
}
```

Repair

Replace 'startsWith()' with 'charAt()'.

```
package OPT;
public class PCTS {
    private void method(String s) {
        if ('a' == s.charAt(0)) {
            // ...
        }
    }
}
```

Reference

Bulka, D. *Java Performance and Scalability Volume 1: Server-Side Programming Techniques*.

OPT.SB

Reserve 'StringBuffer' capacity

Description

This rule flags code that does not provide an initial capacity for ‘StringBuffer’.

The 'StringBuffer' constructor will create a character array of a default size, typically 16. If 'StringBuffer' exceeds its capacity, 'StringBuffer' has to allocate a new character array with a larger capacity, copy the old contents into the new array, and eventually discard the old array. In many situations, you can tell in advance how large your 'StringBuffer' is likely to be. In that case, reserve enough capacity during construction and prevent the 'StringBuffer' from ever needing expansion.

Example

```
package OPT;

public class RSBC {
    void method () {
        StringBuffer buffer = new StringBuffer(); // violation
        buffer.append ("hello");
    }
}
```

Repair

Provide initial capacity for 'StringBuffer'.

```
public class RSBC {
    void method () {
        StringBuffer buffer = new StringBuffer(MAX);
        buffer.append ("hello");
    }
    private final int MAX = 100;
```

```
}
```

Reference

Bulka, D. *Java Performance and Scalability Volume 1: Server-Side Programming Techniques*. Addison Wesley, pp. 30-31.

OPT.SDIV

Use the shift operator on ’a / b’ expressions

Description

This rule flags any ‘a/b’ expression.

"/" is an expensive operation; using the shift operator is faster and more efficient.

Example

```
package OPT;
public class SDIV {
    public static final int NUM = 16;
    public void calculate(int a) {
        int div = a / 4; // should be replaced by "a >> 2".
        int div2 = a/ 8; // should be replaced by "a >> 3".
        int temp = a / 3;
    }
}
```

Repair

```
package OPT;
public class SDIV {
    public static final int NUM = 16;
    public void calculate(int a) {
        int div = a >> 2;
        int div2 = 8 >> 3;
        int temp = a / 3; // not possible to replace this with shift
                                //operator.
    }
}
```

OPT.SMUL

Use the shift operator on ’a * b’ expressions

Description

This rule flags any ‘a*b’ expression.

"*" is an expensive operation; using the shift operator is faster and more efficient.

Example

```
package OPT;

public class SMUL {
    public void calculate(int a) {
        int mul = a * 4; // should be replaced with "a << 2".
        int mul2 = 8 * a; // should be replaced with "a << 3".
        int temp = a * 3;
    }
}
```

Repair

```
package OPT;
public class SMUL {
    public void calculate(int a) {
        int mul = a << 2;
        int mul2 = a << 3;
       int temp = a * 3;       // not possible to replace this with
shift operator.
    }
}
```

OPT.STR

Use ’ ’ instead of " " for one character string concatenation

Description

This rule flags code that uses ““ for string concatenation with one character.

Using ‘’ instead of ““ for one character string concatenation will improve performance.

Example

```
package OPT;
public class STR {
    public void method(String s) {
        String string = s + "d"  // violation.
        string = "abc" + "d"      // violation.
    }
}
```

Repair

Replace one character String Constant with ’ ’.

```
package OPT;
public class STR {
    public void method(String s) {
        String string = s + ’d’
        string = "abc" + ’d’
    }
}
```

OPT.SYN

Never call a "synchronized" method in a loop

Description

This rule flags any “synchronized” method that is invoked inside a loop.

Invoking a "synchronized" method is expensive. Thus, they should not be invoked inside of a loop.

Example

```
package OPT;
import java.util.Vector;
public class SYN {
    public synchronized void method (Object o) {
    }
    private void method () {
        for (int i = 0; i < vector.size(); i++) {
            method (vector.elementAt(i)); // violation
        }
    }
    private Vector vector = new Vector (5, 5);
}
```

Repair

Do not invoke a "synchronized" method in the loop body.

OPT.TRY

Place "try/catch" blocks outside of loops

Description

This rule flags any “try/catch” block inside a loop.

Placing “try/catch” blocks inside loops can slow down code execution.

If “try/catch” blocks are inside loops, code execution can be about 20 percent slower if the JIT compiler is turned off on a JVM without a JIT.

Example

```
package OPT;
import java.io.FileInputStream;

public class TRY {
    void method (FileInputStream fis) {
        for (int i = 0; i < size; i++) {
            try { // violation
                _sum += fis.read();
            } catch (Exception e) {}
        }
    }
    private int _sum;
}
```

Repair

Place “try/catch” block outside of the loop.

```
    void method (FileInputStream fis) {
        try {
            for (int i = 0; i < size; i++) {
                _sum += fis.read();
```

```
            }
        } catch (Exception e) {}
    }
```

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.81 - 83.

OPT.UEQ

Avoid unnecessary equality operations with booleans

Description

This rule flags any unnecessary equality operation with booleans.

Comparing a boolean value with true is an identity operation (returns the same boolean value). The advantages of removing this unnecessary comparison with true are:

1. The code will perform faster (the code generated has about 5 bytecodes less).
2. The code becomes clearer.

Example

```
package OPT;

public class UEQ
{
    boolean method (String string) {
        return string.endsWith ("a") == true;   // Violation
    }
}
```

Repair

```
class UEQ_fixed
{
    boolean method (String string) {
        return string.endsWith ("a");
    }
}
```

OPT.UISO

Avoid unnecessary "instanceof" evaluations

Description

This rule flags unnecessary “instanceof” evaluations.

If the “static” type of the left-hand-side is already the same type as the right-hand-side, then use of the "instanceof" expression is always true.

Example

```
package OPT;

public class UISO {
    public UISO () {}
}

class Dog extends UISO {
    void method (Dog dog, UISO u) {
        Dog d = dog;
        if (d instanceof UISO) // always true.
            System.out.println("Dog is a UISO");
        UISO uiso = u;
        if (uiso instanceof Object) // always true.
            System.out.println("uiso is an Object");
    }
}
```

Repair

Remove the unnecessary "instanceof" evaluations:

```
class Dog extends UISO {
    void method () {
        Dog d;
        System.out.println ("Dog is an UISO");
```

```
    System.out.println ("UISO is an UISO");
}
```

OPT.UNC

Avoid unnecessary casting

Description

This rule flags unnecessary casting.

All classes either directly or indirectly inherit from the class Object and any subclass is implicitly the same type as its superclass. Therefore, cast operations to superclass are not necessary.

Example

```
package OPT;

class UNC {
    String _id = "UNC";
}
class Dog extends UNC {
    void method () {
        Dog dog = new Dog ();
        UNC animal = (UNC)dog;  // unnecessary.
        Object o = (Object)dog;         // unnecessary.
    }
}
```

Repair

```
class Dog extends UNC {
    void method () {
        Dog dog = new Dog();
        UNC animal = dog;
        Object o = dog;
    }
}
```

Reference

Warren, Nigel, and Bishop, Philip. *Java in Practice.* Addison-Wesley, 1999, pp.22-23.

OPT.USB

Use ‘StringBuffer’ instead of ‘String’ for non-constant strings

Description

This rule flags code that uses ‘String’ instead of ‘StringBuffer’ for non-constant strings.

Although the += operator is provided by the ‘String’ class, it is much less efficient than the ‘StringBuffer.append()’ function. Using ‘StringBuffer’ instead of using “+=” on ‘String’ objects will improve performance.

Example

```
package OPT;

class USB {
   public String foobar() {
       String fruit = "apples";
       fruit+= ", bananas";             // Violation
       return fruit;
   }
}
```

Repair

```
public String betterFoobar() {
       StringBuffer fruit = new StringBuffer("apples");
       fruit.append(", bananas");
       return fruit.toString();
   }
```

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.107 - 109

OPT.USC

Use ’String’ instead of ’StringBuffer’ for constant strings

Description

This rule flags code where ‘StringBuffer’ is used for a constant string.

Dynamically resizeable strings are not necessary for constant strings.

Example

```
package OPT;

public class USC {
    String method () {
        StringBuffer s = new StringBuffer ("Hello");
        String t = s + "World!";
        return t;
    }
}
```

Repair

Replace ‘StringBuffer’ with ‘String’ if it is certain that the object will not change. This will reduce the overhead and improve performance.

OPT.UST

Use StringTokenizer instead of 'indexOf()' or 'substring()'

Description

This rule flags code that uses ‘indexOf()’ or ‘substring()’ instead of String-Tokenizer.

String parsing is commonly performed in many applications. Using 'indexOf()' and 'substring()' to parse a String can cause a StringIndex-OutOfBoundsException. The StringTokenizer class makes String parsing bit easier, and is still quite efficient.

Example

```
package OPT;

public class UST {
    void parseString(String string) {
        int index = 0;
        while ((index = string.indexOf(".", index)) != -1) {
            System.out.println (string.substring(index,
string.length()));
        }
    }
}
```

Reference

Larman, G, Guthrie, R *Java 2 Performance and Idiom Guide*. Prentice Hall, 1999, 248 - 249.

OPT.USV

Use ‘stack’ variables whenever possible

Description

This rule flags any non-stack variable that is going to be accessed frequently.

When variables are accessed frequently, consider where these variables are accessed from. Is the variable static, local, or an instance variable? It is about two to three times slower to access “static”, instance variables than to access stack variables.

Example

```
package OPT;
public class USV {
    void getSum (int[] values) {
        for (int i=0; i < value.length; i++) {
            _sum += value[i];           // violation.
        }
    }
    void getSum2 (int[] values) {
        for (int i=0; i < value.length; i++) {
            _staticSum += value[i];
        }
    }
    private int _sum;
    private static int _staticSum;
}
```

Repair

Use stack variables whenever possible if variables are going to be accessed frequently.

You can replace the above "getSum()" method in the following way:

```
void getSum (int[] values) {
    int sum = _sum;  // temporary stack variable.
    for (int i=0; i < value.length; i++) {
        sum += value[i];
    }
    _sum = sum;
}
```

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.122 - 125.

PB.ADE

Avoid dangling “else” statements

Description

This rule flags any dangling "else" statement.

Example

The developer that wrote the following code intended to decrement i if i <= 5. The compiler always associates an "else" with the previous “if” unless instructed by braces to do otherwise. In this example, the "else" is associated with the second “if”; therefore, the decrementation takes place “if (i >= 2)”. To force the structure to execute as originally planned, use braces to indicate to the compiler that the "else" matches the first “if”.

```
package PB;

public class ADE {
    void method () {
        int i = 5;
        if (i < 5)
            if (i < 2)
                i++;
        else
            i--;
    }
}
```

Repair

Include the first “if” structure between braces. The compiler will know that the second “if” structure is the only statement within the first “if” block and the "else" matches the correct “if”.

```
void method () {
        int i = 5;
        if (i < 5) {
            if (i < 2)
                i++;
        } else
            i--;
    }
}
```

PB.AECB

Avoid “catch” blocks with empty bodies

Description

This rule flags any “catch” block that has an empty body.

It is always a good idea to insert error handling code inside of each “catch” block.

Example

```
package PB;
public class AECB {
    public void openFile (String s) {
        try {
        } catch (Exception e) {
           // nothing is done for this exception.
        }
    }
}
```

Repair

Add exception handling code inside of the "catch" block.

PB.ASI

Avoid assignments within an “if” condition

Description

This rule flags any assignment in “if” conditional statements.

Example

```
package PB;

public class ASI {
  public int foo(boolean b) {
     int ret = 0;
     if ((b = true) {    // Violation
        ret = 3;
     }
     return ret;
  }
}
```

Repair

Replace the “=” with “==” or move the variable assignment outside of the “if”.

Reference

http://g.oswego.edu/dl/html/javaCodingStd.html

PB.AUO

Avoid using an object to access “static” fields or methods

Description

This rule flags any “static” field or method that is accessed through an object.

All “static” members should be referenced through a class name.

Example

```
package PB;

class AUO {
    static void staticMethod () {}
    void method () {}

    public static void main (String[] args) {
        AUO object = new AUO ();
        object.staticMethod ();  // Violation
        object.method ();
    }
}
```

Repair

Access “static” methods and fields through the “class”.

```
class AUO_fixed {
    // ...
public static void main (String[] args) {
        AUO object = new AUO ();
        AUO.staticMethod ();
        object.method ();
    }
}
```

PB.CLP

Don't cast primitive datatypes to lower precision

Description

This rule flags any primitive datatype that is cast to lower precision.

By casting to lower precision, the value is truncated and will fall into a different range.

Example

```
package PB;
public class CLP {
    void method () {
        double d = 4.25;
        int i;
i = this.square ((int) d);
    }

    int square (int i) {
        return (i * i);
    }
}
```

Repair

Either explicitly test the value before performing a cast or avoid it.

PB.CTOR

Avoid package-private classes with “public” constructors

Description

This rule flags any “package-private” class with a “public” constructor.

A "public" constructor of a non-"public" class does not have "public" accessibility.Thus, the "public" modifier should be removed because it has no effect and it is confusing.

Example

```
package PB;
class CTOR {    // package accessible class
    public CTOR () {   // public constructor.
    }
}
```

Repair

Change the constructor’s modifier to either “package” or “private”.

PB.DCF

Do not compare floating point types

Description

This rule flags code that compares floating point types.

Comparing floating point numbers for equality can cause logic errors or infinite loops due to unexpected results.

Example

```
package PB;

public class DCF {
    int method (double d) {
        if (d == 1) {
            return 1;
        } else if (d != 2) {
            return 2;
        } else return 3;
    }
}
```

Repair

Use the comparison of floating point numbers sparingly.

PB.DCP

Do not confuse the “+” operator for String concatenation

Description

This rule flags code where you might be confusing the “+” operator for ‘String’ concatenation.

Example

The code below shows the trap you might fall into when using “+” the wrong way. The output of the following example is "2+9 = 29" not "2+9 = 11" as you might have expected.

```
package PB;

public class DCP {
    public static void main (String args []) {
        System.err.println ("2 + 9 = " + 2 + 9);
    }
}
```

Repair

Calculate the result outside the System.out.println statement and print it, or enclose "2+9" in parentheses, i.e: "2+9 = " +(2+9)

PB.DNCSS

Do not call ’setSize()’ in ’ComponentListener. componentResized()’

Description

This rule flags code that calls ’setSize()’ inside of the ’componentResized()’ method’s body.

The “componentResized()” method gets called when the component’s size changes. Invoking the “setSize()” method from within the “componentResized()” method can cause a non-ending sequence of resizing events:

1. User resizes component.
2. “componentResized()” gets invoked.
3. “componentResized()” invokes “setSize()”.
4. “setSize()” posts a component resized event.
5. “componentResized()” gets invoked.
6. ...

Example

```
package PB;

import java.awt.*;
import java.awt.event.ComponentEvent;

public class DNCSS extends Component {
    public void componentResized (ComponentEvent e) {
        Dimension d = getSize();
        setSize(d.width -10, d.height - 10);
                    // causes recursive calls.
```

```
    }
}
```

Repair

Do not call the “setSize()” method inside of the “componentResized()” method’s body.

PB.EQL

Use ’getClass()’ in ’equals()’ method implementation

Description

This rule flags code where an ‘equals()’ method does not use 'getClass()' in the implementation.

Allowing only objects of the same class to be considered equal is a clean and simple solution to implementing the 'equals()' method correctly. The 'getClass()' method returns the runtime class of an object.Therefore, 'get-Class()' can be used to implement the 'equals()' method for the object.

Example

```
package PB;

public class EQL {
    public boolean equals (Object o) {
        super.equals(o);     // violation
    }
}
```

Repair

Use the ‘getClass()’ method before comparing two objects’ equality.

```
    public boolean equal (Object o) {
        if (getClass() != o.getClass())
            return false;
        // ...
    }
```

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp. 44 -47.

PB.EQL2

Use 'instanceof' within an 'equals()' method implementation

Description

This rule flags code where an ‘equals()’ method does not use 'instanceof' in the implementation

Allowing only objects of the same class to be considered equal is a clean and simple solution to implementing the 'equals()' method correctly. The 'instanceof' operator checks if the argument is of the correct type or not. Therefore, 'instanceof' can be used to implement the 'equals()' method for an object.

Note: Jtest’s PB.EQL rule describes another way to implement the 'equals()' method. Because we have seen arguments for both guidelines, we have included both rules and will allow you to decide which one to use.

Example

```
package PB;

public class EQL2 {
    public boolean equals (Object o) {
        super.equals(o);     // violation
    }
}
```

Repair

Use 'instanceof' operator before comparing two objects' equality.

```
public boolean equal (Object o) {
        if (!(o instanceof EQL2))
            return false;
```

```
        // ...
    }
```

Reference

Bloch, Joshua. *Effective Java Programming Language Guide.* Addison Wesley, 2001, pp 25- 34.

PB.FEB

Avoid “for” statements with empty bodies

Description

This rule flags any “for” statement with an empty body.

“for” statements that are immediately followed by a closing statement (for example, a semicolon) are usually typos.

Example

```
package PB;

public class FEB {
    void method () {
        int i;
        for (i = 0; i < 10; ++i) ;
              System.out.println (i);
    }
}
```

The ‘println()’ will only be executed once when i==11.

Repair

Remove the unnecessary semicolon.

PB.FLVA

Do not assign to loop control variables in the body of a "for" loop

Description

This rule flags assignment to a loop control variable in the body of a "for" loop.

A “for” loop control variable should only be modified in the initialization and condition expressions of the "for" loop statement. Modifying them inside the body of the "for" loop makes the loop condition difficult to understand and points to a possible logical flaw in the code.

Example

```
package PB;

public class FLVA
{
    void method1() {
        for (int i = 0; i < 100; i++) {  // Violation
            i += 3;
        }
        for (int i = 0; i < 100; i++) {  // Violation
            i++;
        }
    }
}
```

Repair

Rewrite the code so that the “for” loop control variable doesn’t need to be assigned within the loop body.

PB.IEB

Avoid “if” statements with empty bodies

Description

This rule flags any “if” statement with an empty body.

“if” statements that are immediately followed by closing statements (for example, a semicolon) are usually typos.

Example

```
package PB;

public class IEB {
    void method (int i) {
          if (i < 0) ;
                    System.out.println(“negative i”);
    }
}
```

The ‘println()’ statement will always be executed regardless of the value of ‘i’.

Repair

Remove the unnecessary semicolon.

PB.IMO

Make sure the intended method is overridden

Description

This rule checks for possible typos that may have occurred when overriding methods were written.

Example

```
package PB;

public class IMO {
    protected void finallize () // typo; should be "finalize"
        throws Throwable
    {
        // important cleanup code
    }
}
```

Repair

Fix the typo, or suppress the error message. If the overriding method is spelled incorrectly, it will not be called; the method from the superclass will be called instead.

In the example above, the “important cleanup code” will never be executed.

PB.MAIN

Use the method name 'main' only for the entry point method

Description

This rule checks whether the method name ‘main’ is used for something other than the entry point method.

Because the method name ‘main’ has a special meaning in Java, you can avoid confusion by not using it for purposes other than defining 'public static void main(java.lang.String[])'.

Example

```
package PB;

public class AMOP {

    public static void main(String[] args) {
        System.out.println("This is main method");
    }

    public static void main() {  //violation
        System.out.println("This is another main method");
    }
}
```

Repair

```
package PB;

public class AMOP {

    public static void main(String[] args) {
        System.out.println("This is main method");
    }
```

```
    public static void other_main() {
        System.out.println("This is another main method");
    }
}
```

PB.MPC

Avoid using method parameter names that conflict with class member names

Description

This rule flags any method parameter name that conflicts with a class member name.

Example

The following example shows the use of the same names ‘i’ and ‘j’ for parameters, an instance variable, and a method, respectively. This might create confusion when using the variable or calling the function, and eventually lead to a logic error.

```
package PB;

public class MPC {
    void method (int i, int j) {}
    void j () {}
    private int i = 3;
}
```

Repair

Use different names for parameters and class members (variables and methods) to avoid confusion.

PB.MRUN

The Thread class is missing a ’run()’ method

Description

This rule flags any thread class that does not have a ‘run()’ method.

Classes that extend ‘Thread’ should always have a ‘run()’ method. If a ‘run()’ method is not implemented, this class will not run as multi-threaded.

Example

```
package PB;
public class MRUN {
    public MRUN () {}
    // violation, no run method!
}
```

Repair

Provide a ‘run()’ method for this class.

PB.NAMING

The method name should not be same the class name unless it is a constructor

Description

This rule flags code where a non-constructor method has the same name as its class.

A non-constructor method should not have the same name as its class; if it does, it probably indicates a typo.

Example

```
package PB;

public class NAMING {
    public NAMING () {}   // constructor
    public void NAMING (int size) {}  // not a constructor, it's
          probably a typo.
}
```

Repair

Verify the name of the method and change it as you intended.

Reference

Daconta, M, Monk, E, Keller, J, and Bohnenberger, K. *Java Pitfalls*. John Wiley & Sons, 2000, pp.12 - 14.

PB.NDC

Never define a direct or indirect subclass of class Error, RuntimeException or Throwable

Description

This rule flags code that defines a direct or indirect subclass of class Error, RuntimeException, or Throwable.

The class java.lang.Error is meant for covering abnormal Java Virtual Machine conditions only. If you define a direct or indirect subclass of class java.lang.Error, it is implied that the error is also an abnormal Java Virtual Machine condition, which is not the case. Exception handling of java.lang.Error is not checked by the Java compiler, so erroneous exception handling might not be noticed.

Exceptions in java.lang.RuntimeException and its subclasses are used for avoidable exceptions. The Java compiler does not check the correct handling of these exceptions, so erroneous exception handling might not be noticed.

The class java.lang.Throwable is the super class of java.lang.Exception and java.lang.Error. User-defined exceptions should always be defined as subclasses of java.lang.Exception.

Example

```
package PB;

public class NDC_Exception extends RuntimeException { // violation

    public NDC_Exception (String s) {
        super(s);
    }
}
```

Repair

```
package PB;

public class NDC_Exception extends Exception {

    public NDC_Exception (String s) {
        super(s);
    }
}
```

Reference

Daconta, M, Monk, E, Keller, J, and Bohnenberger, K. *Java Pitfalls*. John Wiley & Sons, 2000, pp.12 - 14.

PB.NEA

Do not use embedded assignment operator

Description

This rule flags code that uses the embedded assignment operator.

Code using embedded assignments becomes cryptic and difficult to read.

Example

```
package PB;

public class PB_NEA {
    void method () {
        int i = 2;
        int j = 2;
        short r = 4;
        double d = 2;
        double x = 3;
        int k = 3;

        d = (k = i + j) + r;
        d -= (x = i + j) + r;
        d /= (x /= i + j) + r;
    }
}
```

Repair

Do not nest assignments; this could lead to confusion. Break the nested assignments into multiple statements.

PB.PDS

Provide “default:” for each “switch” statement

Description

This rule flags any “switch” statement that does not have a “default” label.

Example

```
package PB;

public class PDS {
    void method (int i) {
        switch (i) {
        case 1:
            a = 10;
            break;
        case 2:
        case 3:
            a = 20;
            return;
        }  // missing default label
    }
}
```

Repair

Add a “default” statement.

PB.SBC

Avoid a “switch” statement with a bad “case”

Description

This rule flags code where a missing “break” or “return” causes control to flow into another case in a “switch”.

Example

```
package PB;
public class SBC{
    void method (int i) {
        switch (i) {
        case 1:
             a = 10;
             break;
        case 2:
             a = 20; //missing break
        default:
             a = 40;
             break;
        }
    }
}
```

Repair

Add the missing “break” or “return”.

PB.TLS

Don't use text labels in "switch" statements

Description

This rule flags any text label in a "switch" statement.

Example

The example below shows two situations where errors might occur. Omitting a space between the word “case” and the value being tested prevents the structure from performing the desired action. Also, ‘wronglabel’ will be unused because “switch” handles only “case” labels.

```
package PB;

public class TLS {
    static int method (int i) {
        switch (i) {
        case 4:
        case3:      // i == 3 will not go through here.
            i++;
            break;
        case 25:
        wronglabel: // unused label.
            break;
        }
        return i;
    }

    public static void main (String args[]) {
        int i = method (3);
        System.out.println (i);
    }
}
```

Repair

Change “case3 to “case 3”.

PB.UEI

Use ‘equals()’ when comparing two Objects

Description

This rule flags any case where “==” is used to compare two Objects.

The “==” operator is used to check if two Objects are the same instance of an object, and the '!=' operator used on an Object checks if two Objects are not two identical instances of an object. If you want to check if two Objects have the same value, you should use the ‘equals()’ method.

Example

```
package PB;

import java.awt.*;

public class UEI {
    public boolean CalculateEqual() {
       boolean monthly = co.getSelectedItem() == "Monthly";  // Vio-
lation
       return monthly;
    }
    public boolean CalculateNotEqual() {
       boolean monthly = co.getSelectedItem() != "Monthly";  // Vio-
lation
       return monthly;
    }
    private Choice co = null;
}
```

Repair

Change “==” to ‘equals()’ as follows:

```
package PB;

import java.awt.*;

public class UEI {

    public boolean CalculateEqual() {
       boolean monthly = co.getSelectedItem().equals("Monthly");
       return monthly;
    }
    public boolean CalculateNotEqual() {
       boolean monthly = !(co.getSelectedItem().equals("Monthly"));
       return monthly;
    }
    private Choice co = null;
}
```

Reference

Bloch, Joshua. *Effective Java Programming Language Guide.* Addison Wesley, 2001, pp.25 - 36.

PMETRICS.NB

Number of bytes

Description

This metric measures project size by adding the total number of bytes of all class files in the project.

PMETRICS.NC

Number of classes

Description

This metric measures the total number of classes in the project.

PMETRICS.NJF

Number of Java source files

Description

This metric measures the total number of Java source files in the project.

PMETRICS.NL

Number of lines

Description

This metric measures the total number of lines in the project's source files.

PMETRICS.NOF

Number of fields

Description

This metric measures the total number of fields in a project.

PMETRICS.NOM

Number of methods

Description

This metric measures the total number of methods in a project.

PMETRICS.NPAC

Number of packages

Description

This metric measures the total number of packages in the project.

PMETRICS.NPKGC

Number of package-private classes

Description

This metric measures the total number of package-private classes in the project.

PMETRICS.NPRIC

Number of "private" classes

Description

This metric measures the total number of “private” classes in the project.

PMETRICS.NPROC

Number of "protected" classes

Description

This metric measures the total number of “protected” classes in the project.

PMETRICS.NPUBC

Number of "public" classes

Description

This metric measures the total number of “public” classes in the project.

PORT.ENV

Do not use “System.getenv()”

Description

This rule flags any occurrence of ‘System.getenv()’.

‘System.getenv()’ has been deprecated because it is not portable.

Example

```
package PORT;
public class ENV {
    void method (String name) {
        System.getenv(name);
        java.lang.System.getenv(name);
    }
}
```

Repair

Use ‘System.getProperty()’ and the corresponding ‘getTypeName()’ methods of the “boolean”, “integer”, and “long” primitive types.

Reference

Flanagan, David. *Java in a Nutshell*. O’Reilly, 1999, pp.190-192.

PORT.EXEC

Do not use ‘Runtime.exec()’

Description

This rule flags any occurrence of ‘Runtime.exec()’.

Calling the ‘Runtime.exec()’ method to spawn a process and execute an external command may not be portable because there is no guarantee that the native OS command will behave consistently on different platforms.

Example

```
package PORT;
import java.io.IOException;

public class EXEC {
    public void method(String command) {
        try {
            Runtime.getRuntime().exec(command);
               // violation, not portable
        } catch (IOException io) {
        }
    }
}
```

Reference

Flanagan, David. *Java in a Nutshell*. O’Reilly, 1999, pp.190-192.

PORT.LNSP

Do not hardcode ‘\n’ or ‘\r’ as a line separator

Description

This rule flags code where ‘\n’ or ‘\r’ is hardcoded as a line separator.

Different systems use different characters or sequences of characters as line separators. Therefore, hardcoding ‘\n’, ‘\r’, or ‘\r\n’ damages Java code’s portability.

Example

```
package PORT;

public class LNSP {
    public printHeader (String name, String id) {
        System.out.println(“HEADER \n” +"Name: name \n" +"ID: id
           \n");
    }
}
```

Repair

Use the ‘println()’ method of PrintStream or PrintWriter; this automatically terminates a line with the line separator appropriate for the platform. Or, use the value of System.getProperty(‘line.separator’)

```
package PORT;

public class LNSP {
    public printHeader (String name, String id) {
        System.out.println("HEADER ");
        System.out.println("Name:  " +name);
        System.out.println("ID:  " +id);
    }
```

```
}
```

Reference

Flanagan, David. *Java in a Nutshell.* O'Reilly, 1999, pp. 191-192.

PORT.NATV

Do not use user defined "native" methods

Description

This rule flags any occurrence of a user defined “native” method.

User defined “native” methods are not portable because all “native” methods must be ported to each platform before they become usable.

Example

```
package PORT;

public class NATV {
    native void method (String s); // user defined native method.
}
```

Reference

Flanagan, David. *Java in a Nutshell*. O’Reilly, 1999, pp.190-192.

PORT.PEER

Do not use ‘java.awt.peer.*’ interfaces directly

Description

This rule flags code that uses a ‘java.awt.peer.*’ interface directly.

The interfaces in the ‘java.awt.peer’ are documented as used by AWT implementors only. Applications that use these interfaces directly are not portable.

Example

```
package PORT;
import java.awt.peer.ComponentPeer;
interface PEER extends ComponentPeer
{
    void setName(String name);
}
```

Repair

Do not use ‘java.awt.peer’ package directly.

Reference

Flanagan, David. *Java in a Nutshell*. O’Reilly, 1999, pp.190-192.

SECURITY.CLONE

Make your classes uncloneable

Description

This rule flags cloneable classes.

Java's object cloning mechanism lets you make exact duplicates of objects that have already been instantiated in a running program. This can let an attacker manufacture new instances of classes you define, without executing any of your constructors. If your class is not cloneable, the attacker can define a subclass of your class and make the subclass implement java.lang.Cloneable. This lets the attacker make new instances of your class by copying the memory images of existing objects. This is often an unacceptable way to make a new object.

Repair

If you want your class to be cloneable, you can protect yourself by defining a clone method and making it final. If you're relying on a non-final clone method in one of your super classes, then override the method to make it final or make your entire class final.

Reference

Viaga, J., McGraw,G., Mutsdoch,T, Felten, E.. “Statically Scanning Java Code: Finding Security Vulnerabilities.” *IEEE Software*, September/October 2000.

SECURITY.CMP

Don't compare classes by name

Description

This rule flags code that compares classes by name.

When you want to compare the classes of two objects to see whether they are the same or whether an object has a particular class, you should be aware that there can be multiple classes with the same name in a JVM. A better way is to compare class objects for equality directly.

Reference

Viaga, J., McGraw,G., Mutsdoch,T, Felten, E.. “Statically Scanning Java Code: Finding Security Vulnerabilities.” *IEEE Software*, September/October 2000.

SECURITY.INNER

Do not use inner classes

Description

This rule flags code that uses inner classes.

In Java, it is possible to define inner classes (classes nested inside other classes). Some Java language books say that inner classes can only be accessed by the outer classes that enclose them, but this is false. Java byte code has no concept of inner classes, only regular classes. Consequently, the compiler translates inner classes into ordinary classes that happen to be accessible to any code in the same package.

An inner class can access private variables of the containing class. Because the Java protection mechanism does not let you restrict access to single classes, it must grant access to the entire package. Fortunately, the only variables that are exposed in such a way are those actually used by an inner class.

In addition, a distinction is made between variables that are read by an inner class and those that are written. If an inner class reads a variable, any class in the package can then read that variable. If an inner class writes to a variable, so can any other class in the package.

Example

```
package SECURITY;
>
public class INNER {
    class INNER_Class {  // violation
    }
}
```

Repair

Do not use an inner class unless it is private.

Reference

Viaga, J., McGraw,G., Mutsdoch,T, Felten, E.. “Statically Scanning Java Code: Finding Security Vulnerabilities.” *IEEE Software*, September/October 2000.

SECURITY.PKG

Don't depend on Package Scope

Description

This rule flags code that depends on package-level access.

Package-level access is not secure enough to provide satisfactory security. Java packages are not closed (meaning that new elements can be added to them, even at program runtime). As a result, an attacker can potentially introduce a new class inside your packages and use this new class to access the things you thought you hid.

Repair

Do not rely on package-level access. Make your class, method, field have the least access possible.

Reference

Viaga, J., McGraw,G., Mutsdoch,T, Felten, E.. “Statically Scanning Java Code: Finding Security Vulnerabilities.” *IEEE Software*, September/October 2000.

SECURITY.SER

Make your classes Unserializable

Description

This rule flags Serializable classes.

Java's serialization mechanism lets you save entire objects to a storage mechanism such as a disc, database, or string. The mechanism also lets classes be restored from saved information later, perhaps from the same application after it has stopped and restarted or from another application. Saving an object's state in Java is serialization; restoring its state is deserialization. Serialization is dangerous because it lets adversaries access your objects' internal state. Adversaries can serialize one of your objects into a byte array that can be read, which lets them inspect your object's full internal state, including any fields you marked private and the internal state of any objects your reference.

Repair

Do not make your classes Serializable if possible.

If you make your classes Serializable, make sure that any sensitive data members are "transient."

Reference

Viaga, J., McGraw,G., Mutsdoch,T, Felten, E.. “Statically Scanning Java Code: Finding Security Vulnerabilities.” *IEEE Software*, September/October 2000.

SECURITY.SER2

Avoid making your interfaces Serializable

Description

This rule flags Serializable interfaces.

Java's serialization mechanism lets you save entire objects to a storage mechanism such as a disc, database, or string. The mechanism also lets classes be restored from saved information later, perhaps from the same application after it has stopped and restarted or from another application. Saving an object's state in Java is serialization; restoring its state is deserialization. Serialization is dangerous because it lets adversaries access your objects' internal state. Adversaries can serialize one of your objects into a byte array that can be read, which lets them inspect your object's full internal state, including any fields you marked private and the internal state of any objects your reference.

Repair

If possible, do not make your interfaces Serializable.

If you make your interfaces Serializable, make sure that any sensitive data members are "transient."

Reference

Viaga, J., McGraw,G., Mutsdoch,T, Felten, E.. “Statically Scanning Java Code: Finding Security Vulnerabilities.” *IEEE Software*, September/October 2000.

SERVLET.BINS

Avoid using java.beans.Beans.instantiate ()

Description

This rule flags code that uses java.beans.Beans.instantiate ().

This method will create a new bean instance either by retrieving a serialized version of the bean from disk or by creating a new bean if the serialized form does not exist. The problem, from a performance perspective, is that each time java.beans.Beans.instantiate is called, the file system is checked for a serialized version of the bean. Such disk activity in the critical path of your web request can be costly.To avoid this overhead, simply use "new" to create the instance.

Example

```
package SERVLET;

import javax.servlet.http.*;
import java.beans.*;

public class BINS extends HttpServlet {
    public void doGet (HttpServletRequest rquest)
      throws ClassNotFoundException, java.io.IOException {
        Beans ab = (Beans) Beans.instantiate (
                this.getClass ().getClassLoader (),
"web_prmtv.Bean");
        // do something...
    }
}
```

Repair

Use new someClass () to create a new object instance.

```
public void doGet (HttpServletRequest rquest)
  throws ClassNotFoundException, java.io.IOException {
    Beans ab = new Beans ();
    // do something...
}
```

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

SERVLET.DSLV

Reuse datasources for JDBC connections

Description

This rule flags code that should reuse datasources for JDBC connections, but does not.

A javax.sql.DataSource is obtained from WebSphere Application Server through a JNDI naming lookup. Avoid the overhead of acquiring a javax.sql.DataSource for each SQL access. This is an expensive operation that will severely impact the performance and scalability of the application.

Example

```
package SERVLET;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.sql.*;
import javax.naming.Context;
import javax.naming.InitialContext;

public class DSLV extends HttpServlet {
    public void doGet () throws ServletException {
        DataSource ds = null;    // violation
        try {
            java.util.Hashtable env = new java.util.Hashtable ();
            env.put (Context.INITIAL_CONTEXT_FACTORY, "jndi.
                 CNInitialContext");
            Context ctx = new InitialContext (env);
            ds = (DataSource)ctx.lookup ("jdbc/SAMPLE");
            ctx.close ();
        } catch (Exception e) {
            e.printStackTrace ();
        }
```

```
    }
}
```

Repair

The servlet should acquire the javax.sql.DataSource in the Servlet.init () method (or some other thread-safe method) and maintain it in a common location for reuse.

```
class Better extends HttpServlet {
    // caching the DataSource
    private DataSource ds = null;
    public void init (ServletConfig config) throws ServletException
{
        super.init (config);
        Context ctx = null;
        try {
            java.util.Hashtable env = new java.util.Hashtable ();
            env.put (Context.INITIAL_CONTEXT_FACTORY,
               "jndi.CNInitialContext");
            ctx = new InitialContext (env);
            ds = (DataSource)ctx.lookup ("jdbc/SAMPLE");
            ctx.close ();
        } catch (Exception e) {
            e.printStackTrace ();
        }
    }
}
```

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

SERVLET.HVR

HttpSession variables should be released when finished

Description

This rule flags HttpSession objects that are not released when they are finished.

HttpSession objects live inside the WebSphere servlet engine until:

- The application explicitly and programmatically releases it using the API, javax.servlet.http.HttpSession.invalidate ()
- WebSphere Application Server destroys the allocated HttpSession when it expires (by default, after 1800 seconds or 30 minutes). WebSphere Application can only maintain a certain number of HttpSessions in memory. When this limit is reached, WebSphere Application Server serializes and swaps the allocated HttpSession to disk. In a high volume system, the cost of serializing many abandoned HttpSessions can be quite high.

Example

```
package SERVLET;
import javax.servlet.*;
import javax.servlet.http.*;

public class HVR {
    // violation, no javax.servlet.http.HttpSession.invalidate() is
    //called.
    public void incorrectSession (HttpServletRequest request) {
        HttpSession mySession = request.getSession (false);
        String id = mySession.getId ();
        System.out.println ("HttpSession id = " +id);
    }
}
```

Repair

Call javax.servlet.http.HttpSession.invalidate() when finished.

```
public void correctSession (HttpServletRequest request) {
    HttpSession mySession = request.getSession (false);
    // do something.

    if (mySession != null) {
        mySession.invalidate ();
    }
}
```

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

SERVLET.RRWD

Release JDBC resources when done

Description

This rule flags JDBC resources that are not released when they are finished.

Failing to close and release JDBC connections can cause other users to experience long waits for connections. Although a JDBC connection that is left unclosed will be reaped and returned by WebSphere Application Server after a timeout period, others may have to wait for this to occur.

Close JDBC statements when you are through with them. JDBC ResultSets can be explicitly closed as well. If not explicitly closed, ResultsSets are released when their associated statements are closed.

Example

```
package SERVLET;
import java.sql.*;

public class RRWD {
    void test0 () {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            conn = DriverManager.getConnection ("some url");
            stmt = conn.createStatement();
            rs = stmt.executeQuery ("some query");
        } catch (Exception e) {}
        finally {
            try {
                // should have rs.close () and stmt.close ()
            } catch (Exception e) {}
        }
    }
}
```

Repair

Ensure that your code is structured to close and release JDBC resources in all cases, even in exception and error conditions.

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

SERVLET.SOP

Minimize use of System.out.println or System.err

Description

This rule flags code that uses System.out.println or System.err.

Because System.out.prinln statements and similar constructs synchronize processing for the duration of disk I/O, they can significantly slow throughput.

Example

```
package SERVLET;

import javax.servlet.*;
import javax.servlet.http.*;

public class SOP extends HttpServlet {
    public void service () {
        System.out.println ("starting service");
    }
}
```

Repair

```
public class SOP extends HttpServlet {
    private final static boolean DEBUG_ON = false;
    public void service () {
        // activate tracing only when absolutely needed.
        if (DEBUG_ON) {
            System.out.println ("starting service");
        }
    }
}
```

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

SERVLET.STM

Do not use 'SingleThreadModel' in Servlet class

Description

This rule flags servlet classes that use SingleThreadModel.

SingleThreadModel is a tag interface that a servlet can implement to transfer its re-entrancy problem to the servlet engine. As such, javax.servlet.SingleThreadModel is part of the J2EE specification. The WebSphere servlet engine handles the servlet's re-entrancy problem by creating separate servlet instances for each user. Because this causes a great amount of system overhead, SingleThreadModel should be avoided.

Example

```
package SERVLET;

import javax.servlet.*;
import javax.servlet.http.*;

public class STM extends HttpServlet implements SingleThreadModel {
    // some code.
}
```

Repair

Developers typically use javax.servlet.SingleThreadModel to protect updatable servlet instances in a multithreaded environment. A better approach is to avoid using servlet instance variables that are updated from the servlet's service method.

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

SERVLET.SYN

Minimize synchronization in Servlets

Description

This rule flags excessive synchronization in servlets.

Servlets are multi-threaded. Servlet-based applications have to recognize and handle this. However, if large sections of code are synchronized, an application effectively becomes single threaded, and throughput decreases.

Example

```
package SERVLET;

import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class SYN extends HttpServlet {
    private int numberOfRows = 0;
    private javax.sql.DataSource ds = null;

    public void synExample (HttpServletRequest request) {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement stmt = null;
        int startingRows;

        try {
            synchronized (this) {
                startingRows = numberOfRows;
                String info = null;
                conn = ds.getConnection ("db2admin", "db2admin");
                stmt = conn.prepareStatement ("select * from
                   db2admin.employy");
                rs = stmt.executeQuery ();
                info = rs.getString ("Name");
```

```
            }
        } catch (Exception e) {
        } finally {
             try { rs.close (); }
             catch (Exception e) {}
        }
    }
}
```

Repair

```
public void synBetterExample (HttpServletRequest request) {
        Connection conn = null;
        ResultSet rs = null;
        PreparedStatement stmt = null;
        int startingRows;

        // lock only necessary one.
        synchronized (this) {
            startingRows = numberOfRows;
        }

        try {
            String info = null;
            conn = ds.getConnection ("db2admin", "db2admin");
            stmt = conn.prepareStatement ("select * from
             db2admin.employy");
            rs = stmt.executeQuery ();
            info = rs.getString ("Name");
         } catch (Exception e) {
         } finally {
             try { rs.close (); }
             catch (Exception e) {}
         }
    }
```

Reference

IBM WebSphere Application Server Standard and Advanced Editions, Harvey W. Gunther.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

TRS.ANF

Do not use ’notify()’; use ’notifyAll()’ instead

Description

This rule flags any occurrence of ‘notify()’.

Multiple threads may be waiting on the same object. Using ’notify()’ picks one of the waiting threads and wakes it up. Because there is no way to predict which thread will be awakened, you should use ‘notifyAll()’ to wake up waiting threads.

Example

```
package TRS;
public class ANF {
    public synchronized void notifyThread() {
        notify();
    }
}
```

Repair

Replace ’notify()’ with ’notifyAll()’

Reference

Arnold, Ken, and Gosling, James *The Java Programming Language*. 2d ed. Addison Wesley, 1997, pp.188-190.

TRS.CSFS

Avoid causing deadlock by calling a "synchronized" method from a "synchronized" method

Description

This rule flags code that calls a “synchronized” method from another “synchronized” method.

There are many scenarios for creating deadlocks; most fall into the following categories:

- A thread interdependency in which two or more threads are waiting on one another
- A thread that has an indefinite wait period (such as a blocking call) in which other threads depend on an object that this thread has locked
- A combination of the two.

Example

```
package TRS;
public class CSFS {
    private synchronized void method1 () {
        // do something
    }
    synchronized void method2 () {
        method1 ()  // violation
    }
}
```

Repair

Try not to synchronize the whole method; make a synchronized block that needs to be synchronized.

Reference

Daconta, M., Monk, E., Keller, J., Bohnenberger, K. *Java Pitfalls*. John Wiley & Sons, pp. 50 - 60.

TRS.NSM

Do not use the synchronized modifier

Description

This rule flags any occurrence of the synchronized modifier.

The "synchronized" modifier is equivalent to the "synchronized" statement, but use of only the statement form makes the code easier to understand and debug. However, the "synchronized" modifier may make the code slightly more efficient than the "synchronized" statement.

Example

```
package TRS;

class NSM {
    synchronized void method () {
        // ...
    }
}
```

Repair

Use the “synchronized” statement instead of the “synchronized” modifier, i.e.

```
class NSM_fixed {
    void method () {
        synchronized (this) {
            // ...
        }
    }
}
```

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.122 - 134.

TRS.NSPM

Declare all "public" methods as "synchronized"

Description

This rule flags any “public” method that is not declared as “synchronized”

A "public" method should be synchronized unless it has a Javadoc that describes the assumed invocation context and/or rationale for the lack of synchronization.

In the absence of planning out a set of concurrency control policies, declaring methods as synchronized at least guarantees safety (though not necessarily livens) in concurrent contexts (every Java program is concurrent to at least some minimal extent). With full synchronization of all methods, the methods may lock up, but the object can never enter into randomly inconsistent states (and thus engage incorrect behavior) due to concurrency conflicts. If you are worried about efficiency problems due to synchronization, learn enough about concurrent OO programming to plan out more efficient and/or less deadlock-prone policies (for example, read "Concurrent Programming in Java" by Doug Lea).

Example

```
package TRS;

public class NSPM {
    public void method () {    // violation
        System.out.println("non synchronized public method");
    }
}
```

Reference

http://www.infospheres.caltech.edu/resources/code_standards/recommendations.html

TRS.NSYN

A non-synchronized method should not call 'wait()' or 'notify()'

Description

This rule flags any non-synchronized method that is calling ‘wait()’ or ‘notify()’

Method wait() or notify() is invoked from a method, which is not declared as synchronized. It is not definitely a bug because the monitor can be locked from another method which directly or indirectly invokes the current method.

TRS.RUN

Methods implementing ’Runnable.run()’ should be “synchronized”

Description

This rule flags any method that implements ‘Runnable.run()’ and that is not “synchronized”.

The method ’run()’ of a class that implements the Runnable interface should be synchronized. Multiple threads can be started for the same object implementing the Runnable “interface”; the method ’run()’ can be executed concurrently.

Example

```
package TRS;
public class RUN implements Runnable {
    public void run () {   // violation, this method should be syn-
chronized.
    }
}
```

Repair

Declare ’Runnable.run()’ method “synchronized”.

Reference

http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm

TRS.THRD

Do not call ’Thread.resume()’, ’Thread.stop()’, or ’Thread.suspend()’

Description

This rule flags code that calls ’Thread.resume()’, ’Thread.stop()’, or ’Thread.suspend()’.

’Thread.resume()’, ’Thread.stop()’, or ’Thread.suspend()’ have been deprecated because they are deadlock-prone.

Example

```
package TRS;
public class ThreadStop extends Thread {
    public void method() {
        stop();                      // violation.
    }
}
```

Repair

See the reference.

Reference

See "java.lang.Thread" of API documentation.

TRS.UWNA

Use ’wait()’ and ’notifyAll()’ instead of polling loops

Description

This rule flags any polling loop.

Using 'sleep()' as a polling loop is not efficient because polling loops take up processor cycles to execute the multiple 'sleep()' calls; using 'wait()' and 'notifyAll()' does not.

Example

```
package TRS;
public class UWNA {
    void method (Object o) {
        while (true) {
            while (getStatus()) {
                try {
                    sleep (300);    // violation
                } catch (Exception e) {}
            }
            synchronized (o) {
            // process data.
            }
        }
    }
    boolean getStatus () {
        return _status;
    }
    private boolean _status;
}
```

Repair

Replace "while", and ’sleep()’ with ’wait()’ and ’notifyAll()’.

```
void method (Object o) {
    while (true) {
        synchronized (o) {
            while (getStatus()) {
                try {
                    o.wait ();
                } catch (Exception e) {}
            }
            // process data.
        }
    }
}
```

Reference

Haggar, Peter. *Practical Java - Programming Language Guide*. Addison Wesley, 2000, pp.191 - 194.

TRS.WAIT

The condition test should always be in a loop

Description

This rule flags any condition test that is not inside a loop.

The condition test should always be in a loop. You cannot assume that a thread being awakened indicates that the condition has been satisfied. "if"(cond) statement should never be used for condition test; use "while"(cond) statement.

Example

```
package TRS;
public class WAIT {
    synchronized void method(boolean isWait) {
        if(isWait) {    // should be a "while" statement.
            wait();
        }
    }
}
```

Repair

Replace "if(condition)" with "while(condition)" statement.

Reference

Arnold, Ken and Gosling, James. *The Java Programming Language* 2nd ed. Addison Wesley, 1997, pp.188-190.

UC.AAI

Avoid unnecessary modifiers in an “interface”

Description

This rule flags any unnecessary modifier used in “interface” members.

Interface methods are always “public” and “abstract”. Interface fields are always “public”, “static” and “final”. It is thus unnecessary and confusing to add those modifiers when declaring “interface” fields or methods.

Example

```
package UC;

interface AAI
{
    public void method ();  // Violation
    abstract int getSize ();  // Violation
    static int SIZE = 100;  // Violation
}
```

Repair

```
interface AAI_fixed
{
    void method ();
    int getSize ();
    int SIZE = 100;
}
```

UC.AUV

Avoid unused local variables

Description

This rule flags any unused local variable in your methods.

Example

```
package UC;

public class AUV {
    void method () {
        int i = 4;
    }
}
```

Repair

An unused variable may indicate a logical flaw in the corresponding method. In this case, the method needs to be rewritten to take the unused variable into account.

UC.DIL

Don't explicitly “import” the java.lang.* “package”

Description

This rule flags any case where the java.lang.* “package” is imported.

It is not necessary to import this package because it is imported implicitly.

Example

```
package UC;

import java.lang.*;      // Violation

public class DIL {
}
```

Repair

Remove the import statement for ‘import java.lang’

UC.PF

Avoid unused “private” fields

Description

This rule flags any unused “private” field.

Example

```
package UC;

class PF {
    private int _instanceField;
    private static int _staticField;
}
```

Repair

An unused field may indicate a logical flaw in the corresponding class. In this case, the class needs to be modified to take the unused field into account.

UC.PM

Avoid unused “private” methods

Description

This rule flags any unused “private” method.

Example

```
package UC;

class PM {
    private int unusedMethod () {}
    private static int unusedStaticMethod () {}
}
```

Repair

An unused method may indicate a logical flaw in the corresponding class. In this case, the class needs to be rewritten to take the unused method into account.

UC.UP

Avoid unused parameters

Description

This rule flags any unused method parameter.

Unused parameters waste stack space and cause confusion.

Example

```
package UC;
class UP {
    int findProduct (int x, int y) {  // "y" is unused
        return = x * x;
    }
}
```

Repair

Remove unused parameters or check if the problem is the result of a typo.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z