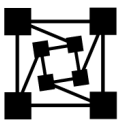
ParaSoft®
jtest®

Automatic Java™ Software and Component Testing:

Using Jtest® to Automate Unit Testing and Coding Standard Enforcement



ParaSoft®

ParaSoft® jtest®

Abstract

For some time now, the development community has been praising such practices as unit testing, coding standard enforcement, metrics measurement, and Design by Contract. When implemented, these techniques can dramatically improve product reliability as well as reduce development time and cost. However, until now, these practices have required so much work that few developers could actually adopt them. Jtest removes this obstacle for Java developers by automating these beneficial techniques. When performing unit testing, Jtest automatically creates and executes test cases that verify class functionality and test class construction; when statically analyzing code, Jtest enforces coding standards that prevent errors and measure metrics that help you flag complicated (and thus error-prone) areas of code. By automating these practices, Jtest makes it easy for even the most time-pressed developers to incorporate them into their development processes and reap the rewards that they offer.

1. Introduction

The key to developing reliable Java™ software on time and on budget is twofold:

- Reduce the opportunity for errors by following Java coding standards.
- Thoroughly test each class as soon as it is developed to prevent small mistakes from growing into widespread, difficult-to-pinpoint problems.

Jtest®, a unique Java unit testing tool from ParaSoft, completely automates both of these tasks so you can perform them as frequently and as thoroughly as needed. Jtest automatically tests any Java class or component without requiring you to write a single test case, harness, or stub. With the click of a button, Jtest automatically tests code construction (white-box testing), tests code functionality (black-box testing), and maintains code integrity (regression testing). No difficult set-up is required; Jtest pinpoints problems immediately. Moreover, if you use Design by Contract™ (DbC) to add specification information to your code, Jtest automatically creates and executes test cases that verify whether a class functions according to its specification. For information on how Jtest leverages DbC information, see our paper “Using Design by Contract to Automate Java Software and Component Testing.”

Jtest also helps you prevent errors with a customizable static analysis feature that lets you automatically enforce over 240 industry-respected coding standards, create and enforce any number of custom coding standards, and tailor the standards enforced to a particular project or group.

This paper explains how development techniques such as unit testing and coding standard enforcement can help you prevent software errors and increase software reliability; in doing so, it describes how Jtest can automate these techniques so that they can become a realistic part of even the most rapid development process.

2. Unit Testing

2.1 What is Unit Testing?

Often, developers hear about unit testing and think the term refers to module testing. In other words, developers think they are performing unit testing when they take a module, or a sub-program that is part of a larger application, and test it. Module testing is important and should certainly be performed, but it is not the technique that we want to concentrate on here. When we use the term “unit testing,” we are talking about testing the smallest possible unit of an application; in terms of Java, unit testing involves testing a class as soon as it is compiled.

2.2 Benefits

Unit testing dramatically improves software quality by helping you detect errors at the stage where it is easiest and most cost-effective to find and fix errors. First of all, because unit testing brings you much closer to the errors, it helps you detect errors that application-level testing might not find. Figures 1 and 2 demonstrate how unit testing does this.

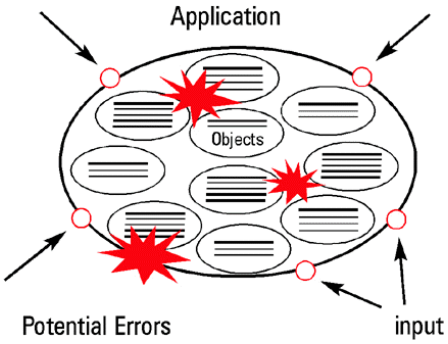


Figure 1: Application Testing

Figure 1 shows a model of testing an application containing many instances of multiple objects. The application is represented by the large oval, and the objects it contains are represented by the smaller ovals. External arrows indicate inputs. Starred regions show potential errors.

To find errors in this model, you need to modify inputs so interactions between objects will force objects to hit the potential errors. This is incredibly difficult. Imagine standing at a pool table with a set of billiard balls in a triangle at the middle of the table, and having to use a cue ball to move the triangle’s center ball into a particular pocket— with one

stroke. This is how difficult it can be to design an input that finds an error within an application. As a result, if you rely only on application testing, you might never reach many of the classes, let alone uncover the errors that they contain.

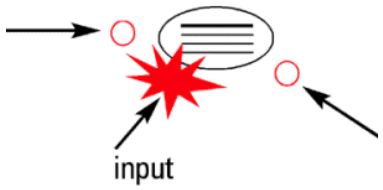


Figure 2: Unit Testing

As Figure 2 illustrates, testing at the unit level offers a more effective way to find errors. When you test one object apart from all other objects, it is much easier to reach potential errors because you are much closer to those errors. The difficulty of reaching the potential errors when the class is tested as an isolated unit is comparable to the difficulty of hitting one billiard ball into a particular pocket with a single stroke.

The second way that unit testing facilitates error detection is by preventing bugs from spawning more bugs, which relieves you from having to wade through problem after problem to remedy what began as a single, simple error. Because bugs build upon and interact with one another, if you leave a bug in your code, chances are it will lead to additional bugs. If you delay testing until the later stages of development, you will probably have to fix more bugs, spend more time finding and fixing each bug, and change more code in order to remove each bug. If you test as you go, it will be easier to find and fix each bug and you'll minimize the chances of bugs spawning more bugs. The result: a significant reduction in debugging time and cost.

2.3 Performing Unit Testing

If performed manually, unit testing tends to be difficult, tedious, and time-consuming. By automating the processes involved, Jtest significantly speeds up unit testing and makes it more thorough and precise.

The first step in performing unit testing is making the class testable. This requires two main actions:

- Designing scaffolding that will run the class.
- Designing stubs that return values for any external resources that are referenced by the class under test, but that are not currently available or accessible.

Creating scaffolding involves creating a new class that can only be used to test the original class. Scaffolding should include the following features:

- A standard way to specify setup and cleanup.
- A method for selecting individual tests or all available tests.
- A means of analyzing output for expected (or unexpected) results.
- A standard form of failure reporting.

If your class references any external resources (such as external files, databases, and CORBA® objects) that are not yet available or accessible, you must then create stubs that return values similar to those that the actual external resource could return. When creating these stubs, you need to choose stub return values that will test the class's functionality and provide thorough coverage of the class.

Several modifications or rewrites might be required to design scaffolding that tests the class thoroughly and accurately. Once the scaffolding is created, you must examine it carefully to ensure that it does not contain any errors. An error in the scaffolding can sabotage the test, but because you cannot test a class in isolation (the original problem), you cannot test the scaffolding either.

After the class is testable, you need to design and execute the necessary test cases. Ideally, you will test the class's construction (i.e., perform white-box testing), test its functionality (i.e., perform black-box testing), then perform regression testing with each modification to ensure that changes did not affect the class's integrity. (These three techniques are described in detail in the sections that follow).

As you can probably see by now, unit testing can consume a fair amount of time, effort, and resources if performed without an automatic unit testing tool; that's why it is rarely performed as often or as thoroughly as it should be. Jtest makes unit testing practical by automating all of the steps involved— even black-box testing. Simply tell Jtest which class or project (a set of classes) you want to test, then Jtest automatically examines each class, generates an appropriate test harness and any necessary stubs, then automatically tests the class using the construction, functionality, and regression testing techniques described below; it also performs static analysis on all available `.java` files (this feature is described in "Coding Standard Enforcement" on page 11).

2.3.1 White-Box (Construction) Testing

White-box (construction) testing validates that unexpected inputs to a class will not cause the program to crash. To perform white-box testing, you design and execute test inputs derived from the class's internal structure 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), as well as if there are coding defects that might make the code more error-prone. The success of white-box testing hinges on the test inputs' ability to cover the class's methods as fully as possible and to find inputs that cause uncaught runtime exceptions.

Preventing and detecting construction problems as early as possible is particularly critical in Java development. In most languages (for example, C and C++), an illegal program operation usually results in the program terminating suddenly. Java, by contrast, provides a very simple way to catch the exceptions that occur at runtime and leave the program running. This mechanism was designed to deal with the checked exceptions (i.e. `java.io` exceptions) to simplify the handling of calls to the underlying operating system and other services. Runtime exceptions, on the other hand, arise from an illegal operation and point to an error in the program. Catching them and letting the program run is usually more problematic than the sudden termination scenario that occurs with C++. The program will keep running and appear as though no problem has occurred, but it will probably enter an inconsistent state, possibly generating incorrect results and/or corrupting the resources that it is accessing.

Although white-box testing is a critical step in ensuring both class and application quality, the difficulty involved in performing white-box testing manually usually causes it to be either skipped or performed less precisely than it should be. Effectively performing white-box testing requires that someone determine exactly what test cases are necessary to fully exercise the class under test. This is incredibly difficult to do manually. Recent studies indicate that a typical company only tests 30 percent of the source code in the programs it develops; the remaining 70 percent is never tested. One reason that so little code is tested is the difficulty of writing test cases that test infrequently executed paths or extreme conditions. Achieving the scope of coverage required for effective white-box testing mandates that a significant number of paths are executed. For example, in a typical 10,000 line program, there are approximately 100 million possible paths; manually generating input that would exercise all of those paths is infeasible and nearly impossible.

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.

For example, let's say you have written the following class and want to test its construction.

```
package examples.eval;

public class Simple
{
    public static int map (int index) {
        switch (index) {
         case 0:
        case10:
            return -1;
        case 2:
         case 20:
        default:
            return -2;
        }
    }

    public static boolean startsWith (String str, String match) {
        for (int i = 0; i < match.length (); ++i)
            if (str.charAt (i) != match.charAt (i))
                return false;
        return true;
    }

    public static int add (int i1, int i2) {
        return i1 + i2;
    }
}
```

Simply tell Jtest where to find this class, then click the Start button. Jtest examines the class, then creates and executes test cases designed to feed it a wide range of unexpected inputs. Jtest's automatically-generated test cases expose the uncaught runtime exception displayed in Figure 3.

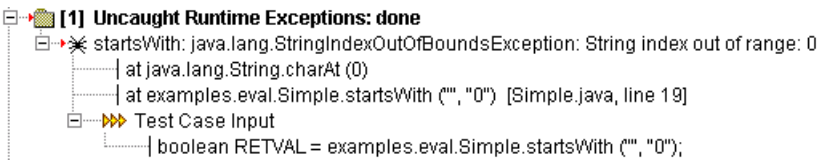


Figure 3: Uncaught runtime exception exposed by Jtest

Figure 4 displays some of the test cases that Jtest automatically created to test this class's construction. These test cases test the class with a wide variety of inputs and fully exercise the class's methods.

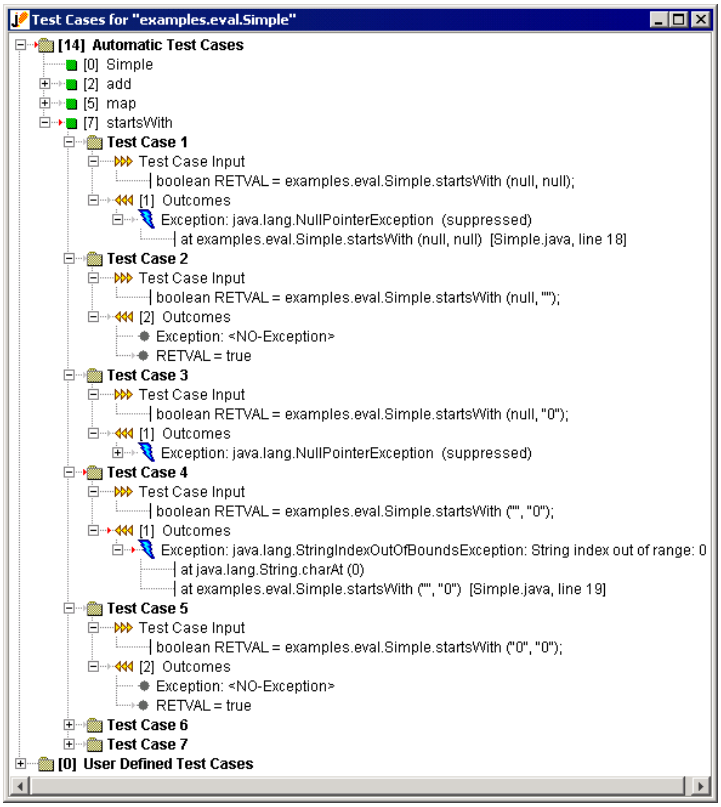


Figure 4: Automatically-created white-box test cases

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 a special `@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.

2.3.2 Black-Box (Functionality) Testing

Black-box (functionality) testing checks that a class behaves according to specification. While it is critical to ensure that a class is constructed strongly, it is equally important to ensure that it does what it is supposed to do, and that all parts of the specification have been fulfilled. To perform black-box testing, you create a set of input/outcome relationships that test whether the class's specifications are correctly implemented. At least one test case should be created for each entry in the specification document; preferably, these test cases should test the various boundary conditions for each entry. After the test suite is ready, you execute the test cases and verify whether the correct outcomes are generated.

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.

Design by Contract (DbC) is a formal way of using comments to incorporate specification information into the code itself. Basically, the code specification is expressed unambiguously using a formal language that describes the code's implicit contracts. These contracts specify such requirements as:

- Conditions that the client must meet before a method is invoked (preconditions).
- Conditions that a method must meet after it executes (postconditions).
- Assertions that a method must satisfy at specific points of its execution.

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.

If any contract violations are found, they are reported under the Jtest UI's Design by Contract Violations branch.

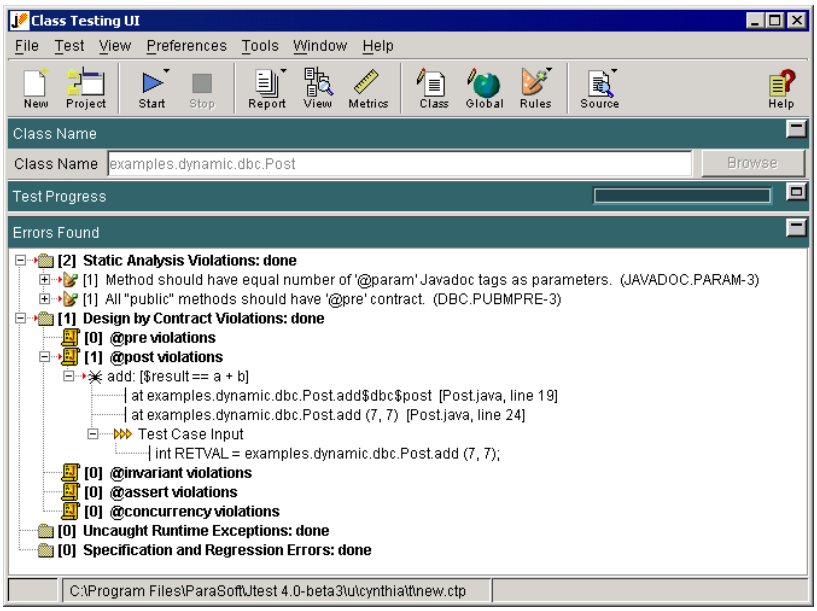


Figure 5: Functionality problem automatically exposed by Jtest

For a more detailed description of how Jtest automatically creates and executes test cases that verify class functionality, as well as information on how DbC information can help focus Jtest's white-box testing, see our paper "Using Design by Contract to Automate Java Software and Component Testing."

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.

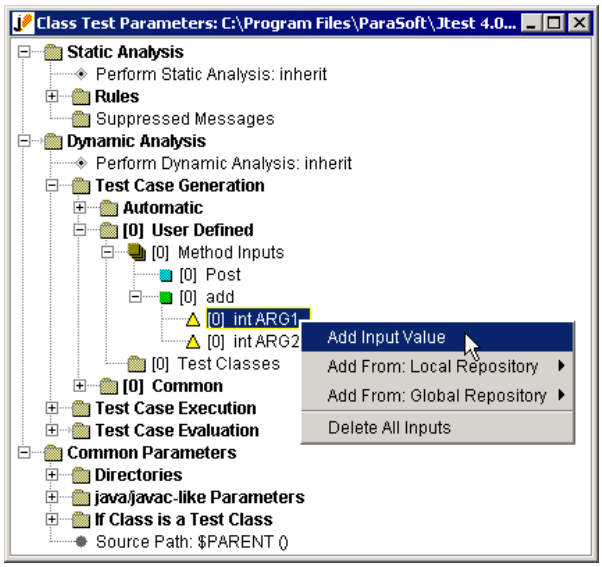


Figure 6: Adding method inputs

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.

2.3.3 Regression Testing

Performing precise regression testing is another necessary step in guaranteeing software quality and reliability. Regression testing— testing modified code under the exact same set of inputs and test parameters used in previous test runs— is the only way to ensure that modifications did not introduce new errors into the class, or to check if modifications successfully eliminated existing errors. Every time a class is modified or used in a new environment, regression testing should be used to check the class's integrity.

Jtest's regression testing feature lets you perform regression testing at the class level; this means that you can run test suites that monitor your code's integrity much earlier than was ever before possible. Jtest completely automates all steps involved in and related to regression testing. Even if you do not specify the correct outcomes, Jtest remembers the outcomes from previous runs, compares them every time the class is tested, then reports an error for any outcome that changes. If you do specify the correct

outcomes, Jtest uses those values as a reference when running regression tests. Whenever Jtest tests a class or set of classes, it automatically saves all test inputs and settings, then adds the test to Jtest's menu options. As a result, all you need to do to perform regression testing is select the appropriate test, then click the Start button. You can also integrate batch-mode Jtest into your nightly builds to ensure that regression errors are always found and fixed as soon as possible.

3. Coding Standard Enforcement

3.1 What Are Coding Standards?

Coding standard enforcement is another software development practice that has been proven to increase application reliability and reduce development time. Coding standards are language-specific "rules" that prevent errors by reducing the opportunity for making errors. Coding standards should be enforced as soon as the code is written and should be implemented in all languages. If they are used consistently, they can prevent entire classes of errors from entering the code.

The best way to explain what coding standards are and how they work is to show an example. In the code below, a simple spacing error destroys the code's functionality:

```
public class PT_TLS {
    static int method (int i) {
        switch (i) {
        case 4:
        case3:
            i++;
            break;
        case 25:
        wronglabel:
            break;
        default:
        }
        return i;
    }

    public static void main (String args[]) {
        int i = method (3);
        System.out.println (i);
    }
}
```

As you can see, the developer intended to write `case 3` but instead wrote `case3`. Because of this simple typographical error, `case3` will now become a text label. Meanwhile, when `i` equals `3`, the value will not go to `case3`. Instead, `i = 3` will always go to the default. This code is not illegal, but it is incorrect.

If the developer of this code had followed the coding standard “Don't use text labels in `switch` statements,” he would have found his mistake and this problem would have been avoided.

3.2 Enforcing Coding Standards

During static analysis, Jtest automatically enforces the above coding standard, as well as over 240 other industry-respected coding standards; this allows you to enforce coding standards without consuming valuable code review time. Jtest statically analyzes each class by parsing the `.java` source and applying a comprehensive set of Java coding standards to it. After analysis is complete, Jtest alerts you to any coding standard violations found.

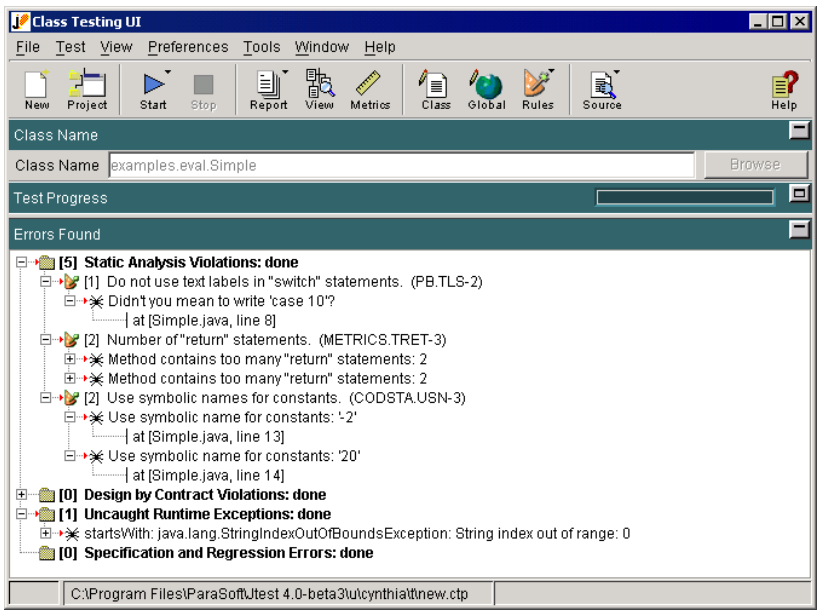


Figure 7: Coding standard violations found by Jtest

Jtest’s coding standards are divided into the following categories:

- Possible Bugs
- Object-Oriented Programming
- Unused Code
- Initialization
- Naming Conventions
- Javadoc Comments

- Portability
- Optimization
- Garbage Collection
- Threads and Synchronization
- Enterprise JavaBeans
- Class Metrics
- Project Metrics
- Miscellaneous
- Internationalization
- Security
- Servlets

In addition, Jtest includes a set of coding standards that help you use Design by Contract (DbC) to add specification information to your code. For more information about this set of coding standards, see our paper "Using Design by Contract to Automate Java Software and Component Testing."

Each of Jtest's coding standards is assigned a violation severity level (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. However, it is simple to tailor Jtest's static analysis feature to meet the needs of a project or development team. With the click of a button, you can enable or disable a single coding standard, or all coding standards that belong to a certain level or category. Because this customization capacity relieves you from having to sort through messages that are not relevant to your team or project, it conserves your time and resources, as well as expedites the error prevention process.

To learn more about how Java coding standards can help you prevent errors, see Adam Kolawa's article, "Coding Standards in Java: Do We Need Them?" in *Java Report*, March 2000 as well as John Viaga and Gary McGraw, Tom Mutsdoch, and Edward W. Felten's "Statically Scanning Java Code: Finding Security Vulnerabilities" in *IEEE Software*, September/October 2000.

3.3 Customized Coding Standards

You can also use the RuleWizard feature to create and enforce customized coding standards that prevent problems that are unique to your coding style, your team, or a certain project. RuleWizard lets you compose and modify coding standards ("rules") by graphi-

cally expressing the pattern that you want Jtest to look for when it parses code during static analysis. Rules are created by pointing and clicking to add rule building blocks to a flowchart-like representation, then using dialog boxes to make any necessary modifications. No knowledge of the parser is required to write or modify a rule. During testing, Jtest implements these custom coding standards along with "built-in" coding standards.

For example, if you found that you repeatedly use assignment in `if` statement condition when you should use equality (i.e., you write `if (a=b)` when you should write `if (a==b)`), you could create and enforce the following coding standard: "Avoid assignment in `if` statement condition."

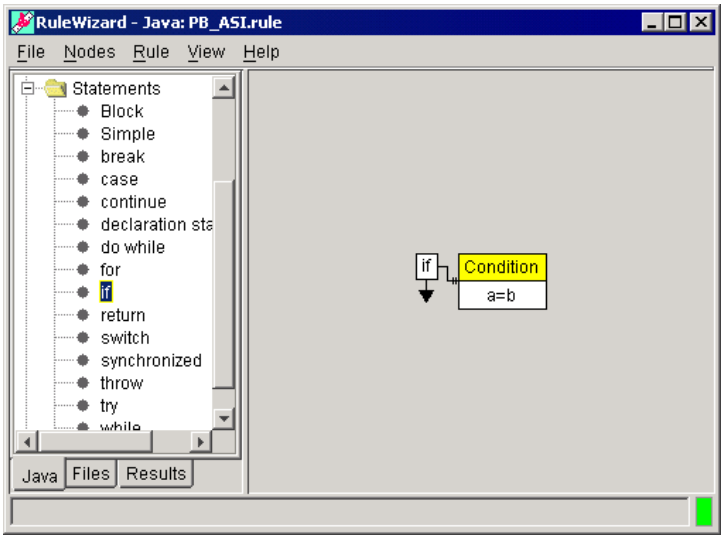


Figure 8: Customized coding standard created with RuleWizard

By providing an easy, flexible way to enforce even the most complex and unique Java coding standards, Jtest helps you perform what many software development experts believe is the most essential task in ensuring software quality: error prevention. Preventing as many errors as possible from entering the code translates not only to less time, effort, and money spent finding and fixing errors as the project progresses, but also to a significantly reduced risk of having errors that elude testing and make their way to the end-user.

3.4 Metrics

A metric is a measurement of a specific attribute or pattern of attributes in a piece of code. For example, a metric might measure the total lines of code in a file, the number of method calls in a class, or the number of return statements in a method. Like traditional

coding standards, metrics measurements can help you prevent errors from entering the code. They do this by indicating which areas of code are most complicated and thus most error-prone and difficult to debug.

For error-prevention purposes, the most effective metrics are specific and correlated to particular areas of code. For example, it might be nice to know that your code is generally complicated, but it is much more useful to know which specific classes and methods are the most complicated and what is causing them to be complicated. If you use metrics analysis to target the most complicated code, you can simplify the code before problems arise.

It is also beneficial to measure more general, project-wide metrics. When metrics are used consistently and are tracked across multiple team or company projects, they can be used to make determinations about project length, cost, and status.

Jtest automatically measures both class and project metrics when it performs static analysis. If any of your metrics are out of the “legal” bounds that we suggest (or that you customize), Jtest will report a static analysis violation for each out-of-bound metric. These messages explain the standard violated and report the exact location of the problem, so you can easily determine what code should be simplified and how it should be changed.

Jtest also offers a summary of all metrics for each class and each project (a project is a set of classes tested at once; for example, all classes in a certain zip file or directory).

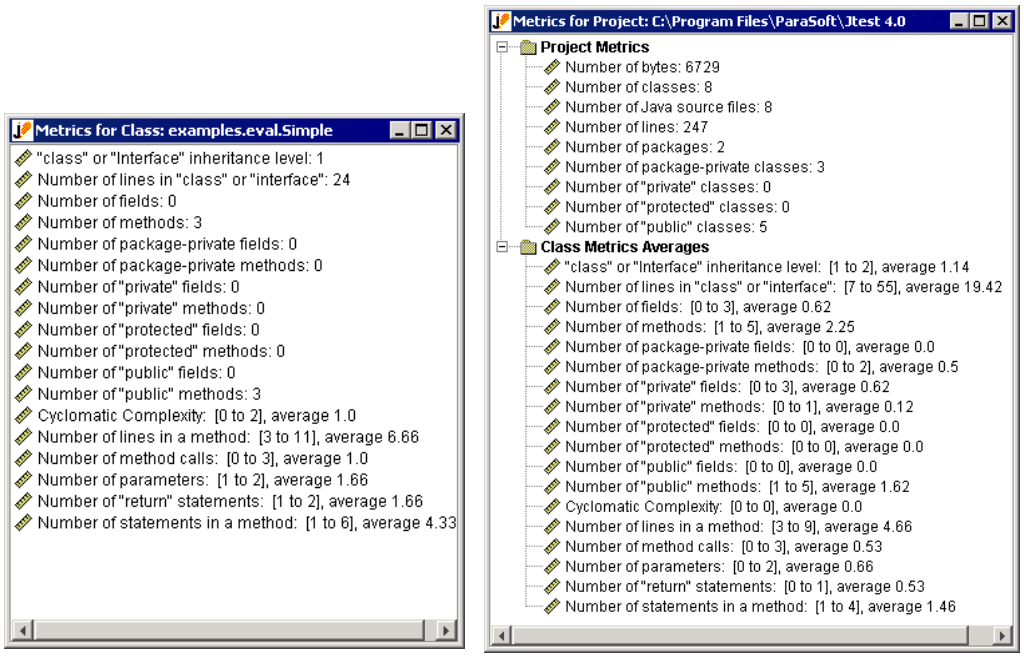


Figure 9: Class and Project Metrics

In addition, Jtest helps you track metrics across the span of a project; it saves your project metrics for each test and can graph how the following metrics change over time:

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

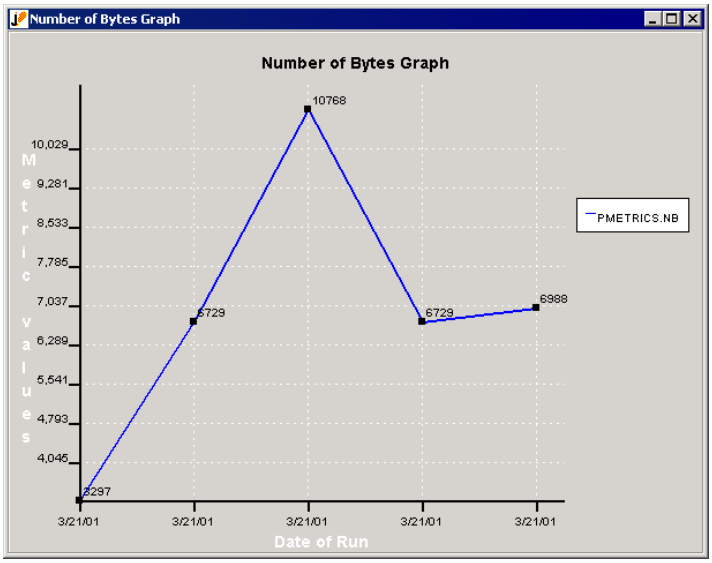


Figure 10: Metrics graph: bytes over time

4. Conclusion

For some time now, the development community has been praising such practices as unit testing, coding standard enforcement, metrics measurement, and Design by Contract. When implemented, these techniques:

- Decrease the number of errors in your code.
- Reduce the amount of debugging you need to perform.
- Improve the quality of the software you release.
- Reduce development and maintenance time and cost.

Until now, these practices have required so much work that few developers could actually adopt them. By automating these practices, Jtest makes it easy for even the most time-pressed developers to incorporate them into their development processes and reap the rewards that they offer.

5. References

Kolawa, A., “Coding Standards in Java: Do We Need Them?” *Java Report*, March 2000.

ParaSoft Corporation, “Using Design by Contract to Automate Java Software and Component Testing.” http://www.parasoft.com/jtest/papers/tech_dbc.htm.

Schroeder, M., “A Practical Guide to Object-Oriented Metrics.” *IT Pro*, November/December 1999.

Viaga, J., McGraw, G., Mutsdoch, T. and Felten, E., “Statically Scanning Java Code: Finding Security Vulnerabilities.” *IEEE Software*, September/October 2000.

6. Availability

Jtest is available now at www.parasoft.com/jtest. To learn more about how Jtest and other ParaSoft development tools can help your department prevent and detect errors, talk to a Software Quality Specialist today at 1-888-305-0041, or visit www.parasoft.com.

7. Contacting ParaSoft

USA

2031 S. Myrtle Ave.
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626)305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 1 64 89 26 00
UK: Tel: +44 171 288 66 00
Germany: Tel: +49 (0) 78 05 95 69 60
Email: info-europe@parasoft.com

Last Updated 3/27/01