

Automatic Java™ Software and Component Testing:

Using Jtest® to Automate Unit Testing and Coding Standard Enforcement

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 and 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 class construction; when statically analyzing code, Jtest enforces coding standards that prevent errors and measure metrics that help pinpoint 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 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 thoroughly as needed. Jtest automatically tests any Java class, JSP, 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 as specified. 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 prevent errors with a customizable static analysis feature that lets you automatically enforce over 300 industry-respected coding standards, create and enforce any number of custom coding standards, and tailor the standards enforced to a particular project or group.

Jtest fits seamlessly into existing development processes and Java development IDEs such as IBM Visual Age, IBM WebSphere Studio Application Developer, Sun Forte, and

Borland JBuilder. It contains special group sharing features so your entire team can easily share Jtest settings and projects. In addition, it integrates with CVS source control so that group file access is managed as effortlessly as possible.

This paper explains how development techniques such as unit testing and coding standard enforcement can help prevent software errors and increase software reliability. Along the way, it describes how Jtest automates these techniques, enabling them to become a realistic part of any development process.

2. Unit Testing

2.1 What is Unit Testing?

Often, developers hear about unit testing and think of 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 when we use the term “unit testing,” we are talking about testing the smallest possible part 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 during implementation, 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.

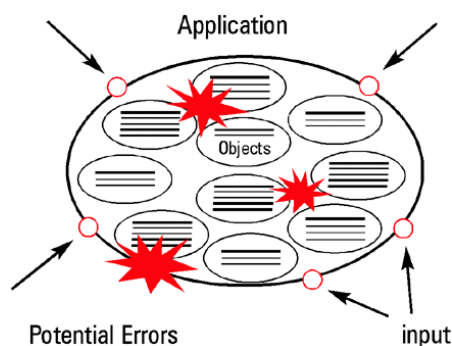


Figure 1: Application Testing

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.

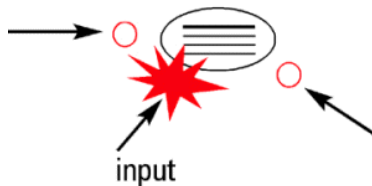


Figure 2: Unit Testing

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 the 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. It's challenging, yet definitely possible.

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 not only spend more time fixing each bug, but you will also have more bugs to fix. If you test as you go, you will avoid this scenario. 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. However, by automating the processes involved, Jtest significantly speeds up unit testing and makes it more thorough and precise. In this section, we cover the general steps that unit testing entails, then describe how Jtest automates them.

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 the actual external resource could return. When creating stubs, you need to choose return values that will test the class's functionality and provide thorough coverage.

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 errors. An error in the scaffolding can sabotage the test. However, because you cannot test a class in isolation (the original problem), you cannot test the scaffolding either.

Once 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 necessary. 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 appropriate scaffolding and any necessary stubs, then automatically tests the class using the construction, functionality, and regression testing techniques described later in this section; 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 case inputs derived from the class's internal structure to determine whether any possible class usages will make the class crash (in Java, this is equivalent to throwing an uncaught runtime exception) and whether coding defects might make the code more error-prone. The success of white-box testing hinges on whether your test case inputs cover the class's methods thoroughly enough to expose its hidden weaknesses.

Preventing and detecting construction problems in the early stages of development is particularly critical in Java. 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 exceptions without stopping program execution. 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. However, a runtime exception arises from an illegal operation and usually points to a critical program error. Catching the exception and letting the program execution continue is usually more problematic than the sudden termination 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 class and application reliability, 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 required to fully exercise the class under test. This is incredibly difficult to do manually. Recent studies indicate that a typical company only tests 30% of the source code in the programs it develops; the remaining 70% 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. To achieve the scope of coverage required for effective white-box testing, you must execute a significant number of paths. For example, in a typical 10,000 line program, there are approximately 100 million possible paths. Manually generating input that would exercise all 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 detected, Jtest reports the exception type, the stack trace, and 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:
            case 10:
                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 inputs. Jtest's automatically-generated test cases expose the uncaught runtime exception displayed in Figure 3.

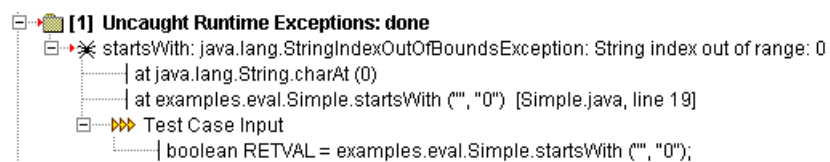


Figure 3: Uncaught runtime exception exposed by Jtest

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

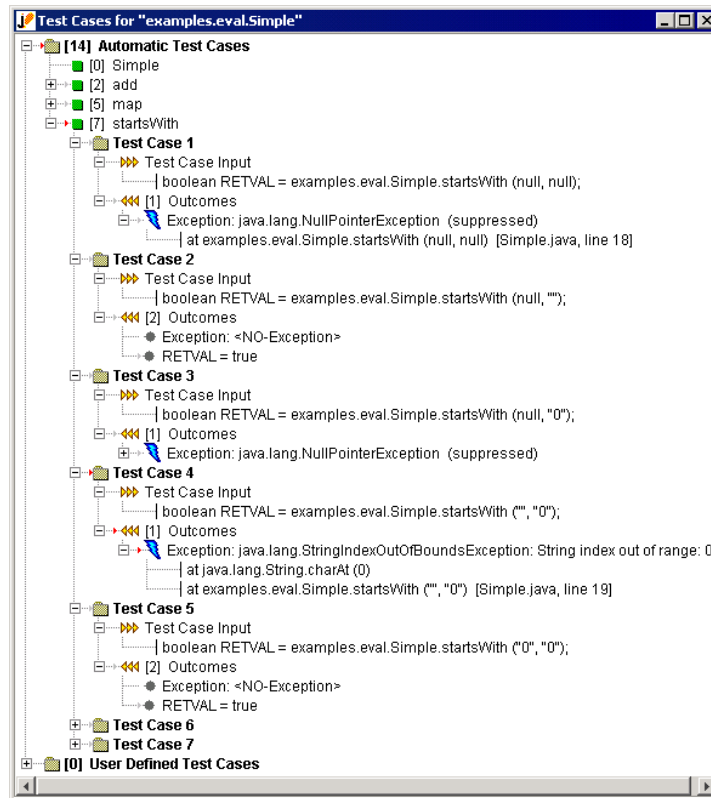


Figure 4: Automatically-created white-box test cases

Jtest can perform white-box testing on any Java class, JSP, 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 verify whether 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 a valid exception in the code using the `@exception` comment tag, Jtest will not report 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.

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 functions as expected, and that all parts of the specification have been fulfilled. To perform black-box testing, you typically create a set of input/outcome relationships that test whether the class's specifications are implemented correctly. 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 classes use Design by Contract (DbC) to express specification information, Jtest completely automates the black-box testing process. If your classes do not use DbC, Jtest makes the black-box testing process significantly easier and more effective than it would be if you were creating test cases on your own.

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 must hold true before a method can execute (preconditions).
- Conditions that must hold true after a method completes (postconditions).
- Conditions that must hold true any time a client can invoke an object's method (invariants).
- Method body assertions that must evaluate to true (assertions).

Jtest reads each class's DbC specification information, then automatically develops test cases based on this specification. Jtest designs its test cases as follows:

- If the code has `@post` contracts, Jtest creates test cases that verify whether the code satisfies those conditions.
- If the code has `@assert` contracts, Jtest creates test cases that try to make the assertions fail.
- If the code has `@invariant` contracts, Jtest creates test cases that try to make the invariant conditions fail.
- If the code has `@pre` contracts, 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 @pre contracts, Jtest determines whether the method under test can pass non-permissible values to the other methods.
- If any class under test (with or without contracts) calls a class that contains contracts, Jtest determines whether the class under test can interact with the second class in a way that violates the contract.

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

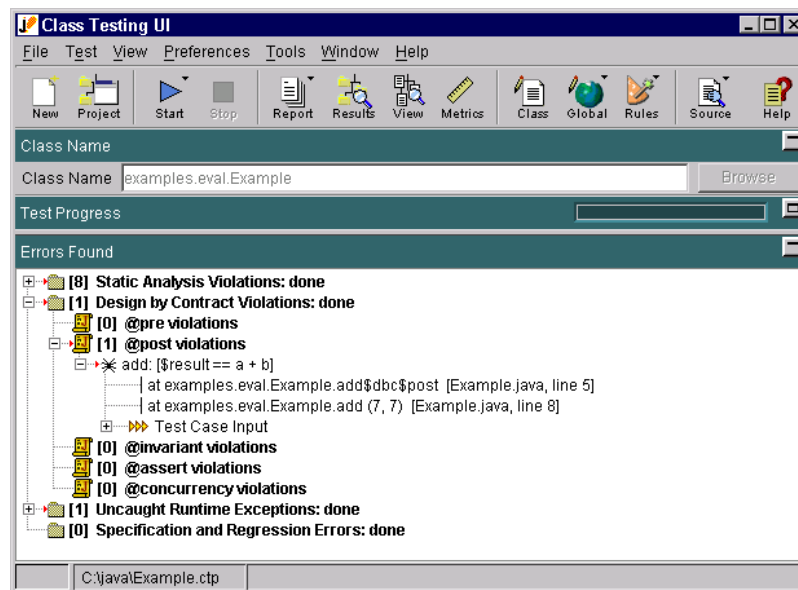


Figure 5: Functionality problem automatically exposed by Jtest

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 DbC. You can use Jtest's automatically-generated set of test cases as the foundation for your black-box test suite, then extend the test suite 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-like Test Classes for test cases that are too complex or difficult to be added as method inputs. JUnit Test Classes can be created automatically based on automatic or user-defined inputs, and JUnit Test Class templates can be automatically generated for any class so you never have to build a Test Class from scratch.

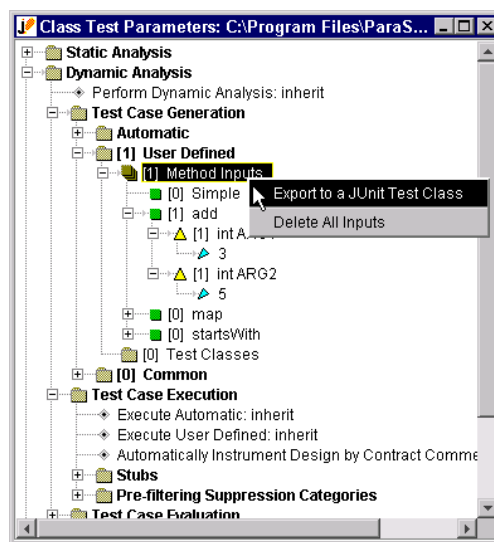


Figure 6: Creating a Test Class based on method inputs

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 or regression testing errors occur on subsequent test runs.

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 modifica-

tions 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 lets you perform regression testing at the class level; this means that you can run test suites that monitor your code's integrity early in the development process. 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 the outcomes every time the class is tested, then reports an error for any outcome that changes. If you 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 applied 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 PB_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 300 additional 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 to it a comprehensive set of Java coding standards. After analysis is complete, Jtest alerts you to any coding standard violations found.

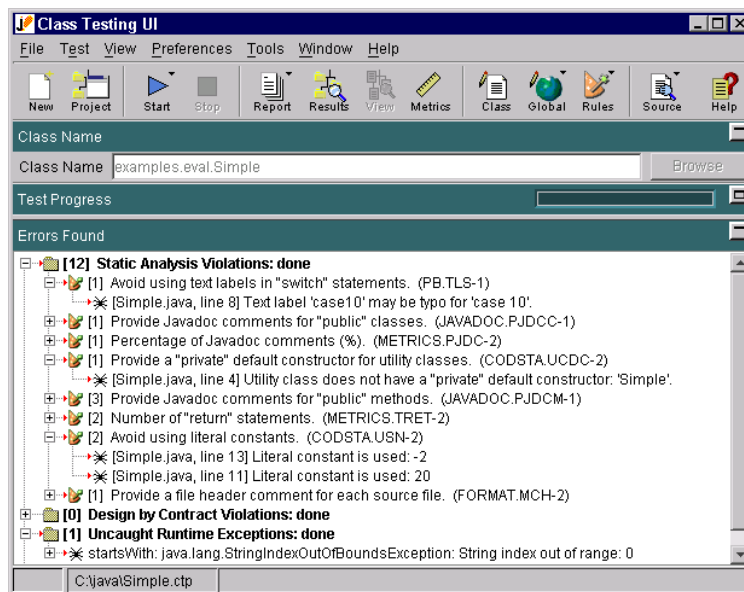


Figure 7: Coding standard violations found by Jtest

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

- Possible Bugs
- Object-Oriented Programming
- Unused Code

- Formatting
- Initialization
- Naming Conventions
- Javadoc Comments
- JUnit Test Cases
- 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 contracts 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 and 2. However, it is easy 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 in a certain level or category. You can also create rule mapping files to modify rule categorization or severity. Because this customization capacity relieves you from having to sort through messages that are not relevant to your team or project, it expedites the error prevention process.

To learn more about how Java coding standards can help prevent errors, see the “Enforcing Coding Standards” chapter in Adam Kolawa et al, *Bulletproofing Web Applications* as well as John Viaga et al, “Statically Scanning Java Code: Finding Security Vulnerabilities” in *IEEE Software*, September/October 2000.

3.3 Customized Coding Standards

If you want to create and enforce customized coding standards that prevent problems unique to your coding style, team, or project, you can do so with Jtest's RuleWizard feature. RuleWizard lets you compose and modify coding standards ("rules") by graphically expressing the pattern you want Jtest to find 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 find 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."

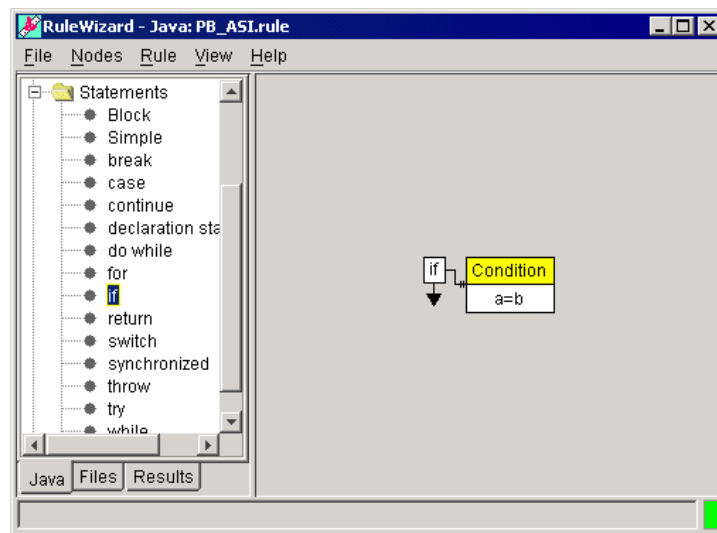


Figure 8: Customized coding standard created with RuleWizard

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, metric measurements can help 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 interesting 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 why. 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 you use metrics consistently and track them across multiple team or company projects, you can use them to make determinations about project length, cost, and status.

Jtest automatically measures both class and project metrics during static analysis. If any metrics fall outside the specified “legal” bounds, Jtest reports a static analysis violation for each out-of-bound metric. These messages 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.

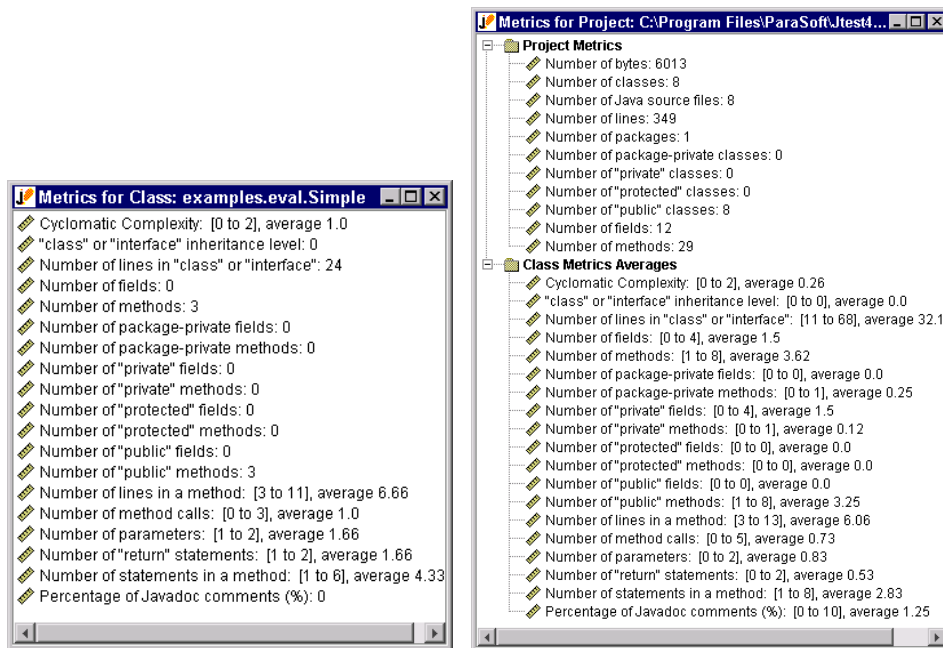


Figure 9: Class and Project Metrics

In addition, Jtest tracks metrics across the duration of a project; it saves project metrics for each test, and graphs 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.

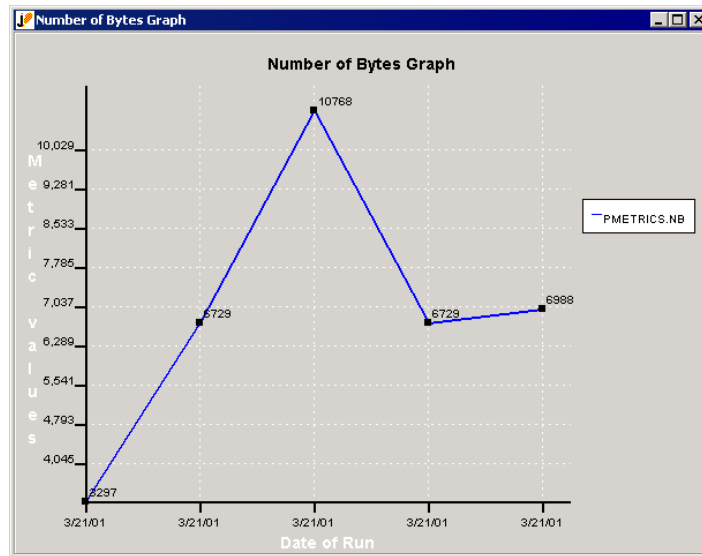


Figure 10: Metrics graph: bytes over time

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 they offer.

5. References

Kolawa, A., Hicken, W., Dunlop, C., *Bulletproofing Web Applications*. Hungry Minds, Inc. (ISBN 0764548662), 2001.

Parasoft Corporation, "Using Design by Contract to Automate Java Software and Component Testing." <http://www2.parasoft.com/jsp/products/article.jsp?articleId=579&product=Jtest>.

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. 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 (020) 8263 2827
Germany: Tel: +49 7805 956 960
Email: info-europe@parasoft.com

Parasoft and Jtest are registered trademarks of Parasoft Corporation. RuleWizard and Jcontract are trademarks of Parasoft Corporation. All other brands are trademarks or registered trademarks of their respective holders.

Last Updated 3/8/02